1994

An Analysis of the Paging Activity of Parallel Programs

Kuei Yu Wang

Dan C. Marinescu

Report Number:
94-042

An Analysis of the Paging Activity of Parallel Programs

Part I. Correlation of the Paging Activity of Individual Node Programs in the SPMD Execution Mode

Kuei Yu Wang[1] and Dan C. Marinescu[2]
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR-94-042
June 1994

AN ANALYSIS OF THE PAGING ACTIVITY OF PARALLEL PROGRAMS

Part I. Correlation of the Paging Activity of Individual Node Programs in the SPMD Execution Mode

Kuei Yu Wang*and Dan C. Marinescu†
Computer Sciences Department
Purdue University
West Lafayette, IN 47908
CSD-TR-94-042

August 3, 1994

Abstract

In this paper we introduce a methodology for the analysis of the paging activity of parallel programs running on massively parallel systems. In the first part of this paper, we study the correlation of the paging activities of individual node programs in the SPMD execution mode and its effect on scheduling. The second part of the paper studies the load placed upon the I/O and the communication systems by the paging activity. The third part describes the tools for monitoring and analysis of the paging behavior of a parallel program.

*Work supported in part by CNPq Brazil
†Work supported in part by NSF under grants CCR-9119388 and BIR-9301210

Contents

1 Introduction

Massively parallel systems (MPPs) are viewed today as expensive scientific and engineering instruments. Their primary use is in the area of numeric simulation of complex physical phenomena. The performance/usability trade-offs of such systems are heavily tilted in favor of performance. Most distributed memory MIMD (DMIMD) systems have rather primitive operating systems with restricted functionality, and rudimentary management of system resources. Only recently MPPs which run standard operating systems in all Processing Elements (PEs) have been announced, e.g., IBM's, SP1 and SP2 (running AIX), and Intel Paragon (running OSF/1 under Mach). Such systems are easier to use but less efficient than their counterparts which run only communication kernels (e.g., SUNMOS, NX, etc.).

Virtual memory is a convenience function supported by operating systems, which allows the user to design his/her application without an immediate concern for the amount of real memory available on a certain system. The operating system maps the virtual (user) address space into the real memory available. If the application exhibits a good locality of reference, then the performance penalty associated with virtual memory is low, even when the virtual address space is considerably larger than the real memory.

The support for virtual memory is an important step towards making massively parallel systems more usable and more appealing for a broader class of applications. Yet existing distributed memory MIMD systems are unbalanced; their I/O and communication bandwidths are insufficient to sustain the request rates generated by powerful processors. There is a legitimate concern that the paging activity may lead to a significant performance penalty by increasing the I/O and the communication load.

The goal of our research is to observe and understand the paging activity of parallel programs. We want to answer questions like: (a) How to characterize the paging activity of a parallel program? (b) How is the paging activity affected by changes in the number of processing nodes and the size of the data space? How does it change when the system configuration changes, e.g. the placement and/or the number of I/O nodes, etc. ? (c) How can the knowledge of the paging activity of an application be used to improve its performance? (d) How can the knowledge of the paging activity of several applications be used to improve the concurrent scheduling of these applications in different partitions of a large system? Such questions can only be answered by studying the paging activity of representative applications running on existing MPPs. Therefore our first objective is to develop a methodology for the study of paging activity which includes program monitoring and the analysis of the collected data. This paper discusses the development of an application paging profiler and a post-profiling tool. The profiler captures the application's paging activity by producing trace records during the execution time. The post-profiling

tool processes and summarizes the large trace record set, creating a concise and representative paging model for the application under study. Visualization tools support graphical representation of raw and processed trace data.

In this paper we report on the paging activity of parallel programs running under OSF/1 on an Intel Paragon XP/S supercomputer. The parallel programs we have profiled and analyzed are structural biology programs used for the determination of the 3-D atomic structure of large macromolecules like viruses [4, 11].

The more significant parameters describing the paging activity of a program running under OSF/1 are: page faults, copy-on-write, page-outs and page-ins. A *page fault* is an event detected by hardware during the address translation when a page entry in the page table of a task is invalid. A *copy-on-write* event occurs when one of the threads sharing a page requests a write access to that page. A *page-out* event occurs when the replacement algorithm decides to reuse the frame containing an inactive page. A *page-in* event occurs when the page currently referenced is not in memory.

In §2 of this paper, we discuss the relationship between paging and scheduling. In uniprocessor systems, context switching is used to hide the high latency associated with a page-in request. Parallel systems use gang scheduling and the strategy mentioned above for hiding the latency of a page-in could only be used if all node programs of an application experience page faults leading to page-in requests (§3.4) precisely at the same time. This motivates our studies of the correlation of the paging activities of all the node programs. Then we introduce a model of the paging activity. In §3, the virtual memory implementation under Mach and the relevant statistical data collected during execution are discussed.

At the present time we are concentrating on a two prong data analysis. First, we study how similar/dissimilar is the paging activity of different node programs in SPMD, Same Program Multiple Data, execution mode by performing a "skyline analysis" of the data collected for different node programs. This analysis is done by determining the rate at which different events occur, by isolating the peaks of activity from the background and by correlating the time of occurrence and the amplitude of these peaks. This skyline analysis for the page faults, page-ins, copy-on-write, and page-outs, is presented in the first part of this paper. The second type of analysis is the "cumulative profile", used to determine the total load due to paging activity upon the communication and the I/O sub-system. The "cumulative profile" for page-ins and page-outs is presented in the second part of this paper.

2 Characterization of the paging activity of parallel programs

A massively parallel system consists of compute nodes, I/O nodes and service nodes connected by a high speed interconnection network. Network topologies used in existing systems are hypercubes (e.g., the Intel iPSC/860, the NCUBE), 2-D meshes (e.g., the Intel Paragon), 2-D tori (e.g., the Cray MPP), fat-trees (e.g., Thinking Machines CM5), or extra stage omega networks (e.g., IBM's SP1 and SP2).

Each compute node consists of one or more processors having a common memory, possibly co-processors (e.g., a message co-processor) and a network interface. In addition to the configuration mentioned above an I/O node has I/O interfaces for devices like disks, and/or computer networks. Space and cost considerations limit the number of I/O nodes, therefore the I/O bandwidth of current systems.

MPPs are partitioned statically; a partition is allocated a number of compute nodes and shares with other partitions the set of I/O nodes. A parallel program runs in a partition of a size determined by the needs of that application. A parallel program consists of a set of node programs, one for each PE.

2.1 Paging and scheduling

The paging behavior of a sequential program (process) is characterized by its *working set* defined as the collection of pages needed for process execution over a period of time. Figure 1 shows a well behaved process (called process A) which exhibits a good locality of reference and has a relatively small working set, w_A in contrast with process B which requires a large working set, w_B.

The execution time in the presence of page faults, denoted by T_f is

$$T_f = T(1 + t^p \times \eta^p) = T(1 + a^p)$$

with

- T – the execution time without any page fault
- η^p – the actual page fault rate, the number of page faults per unit of time
- t^p – the latency of a page fault
- $a^p = t^p \times \eta^p$

The page fault rate depends upon the relationship between the size of the working set of the process and the amount of memory available. The page fault latency varies considerably

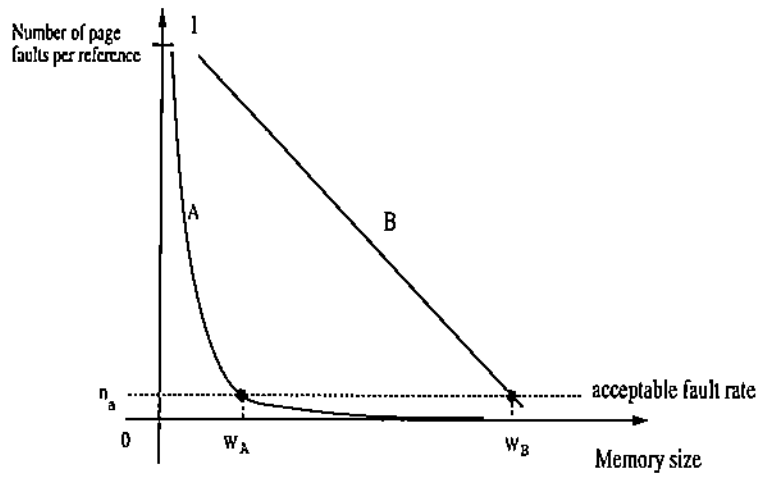


Figure 1: The working set of a process and the acceptable fault rate. Process A has good locality of reference, and the size of its working w_A set is relatively small, while process B has a fairly large working set, w_B because it does not exhibit a good locality of reference.

depending upon the cause of the page fault. Page-ins require the longest time because they involve access to the swap file. In our simplified analysis we are primarily concerned with page faults due to page-ins and we will use the following approximation $\eta^p \cong \eta_f$ with η_f the rate of page-ins and $t^p = t_f$ with t_f the latency of a page-in. In this case, $T_f = T(1+a)$ with $a = t_f \times \eta_f$. In a multiprogramming environment, the operating system hides the latency of a page fault by suspending the process experiencing a page fault and scheduling another process ready to run.

Let us now consider a parallel system with N_c compute nodes and N_{IO} I/O nodes and assume that every page fault requires an I/O operation. We expect the service time for a page fault to increase due to contention for shared resources like communication channels and I/O nodes. To obtain a very crude estimate of t_f^p, the time needed in this case for resolving a page fault leading to a page-in request, we consider a model of the system based upon a set of simplifying assumptions

(a) All compute nodes have identical page-in rates of η_f pages/second. The aggregate page-in rate of the parallel program consisting of N_c identical tasks running concurrently is $N_c \times \eta_f$.

(b) The page-ins are evenly distributed over the set of I/O nodes. Each I/O node runs a pager and has a request rate of

$$\eta_f^p = \frac{N_c}{N_{IO}} \eta_f$$

(c) Each I/O node can be modeled as an M/M/1 system with request rate of η_f^p and service rate of $1/t_f$. Then the time needed for a page-in is

$$t_f^p = \frac{t_f}{1 - t_f \times \eta_f^p}$$

The condition for the system to be stable is

$$t_f \times \eta_f^p = t_f \times \eta_f \times \frac{N_c}{N_{IO}} < 1$$

or

$$\eta_f < \frac{N_{IO}}{N_c} \frac{1}{t_f}.$$

For example if the ratio $n_{IOC} = \frac{N_{IO}}{N_c} = 0.1$ and $\frac{1}{t_f}$ is 20 pages/sec then $n_f \leq 2$ pages/sec.

(d) The communication delays can be neglected.

(e) We assume a linear speed-up namely the parallel execution time, in absence of page faults is:

$$T^p = \frac{T}{N_c}$$

It follows that the parallel execution time in the presence of page faults is

$$T_f^p = T^p(1 + t_f^p \times \eta_f) = \frac{T}{N_c}\left(1 + \frac{t_f \times \eta_f}{1 - \frac{N_c}{N_{IO}} \times t_f \times \eta_f}\right) = \frac{T}{N_c}\left(1 + \frac{a}{1 - a \times n_{IOC}}\right)$$

This approximate analysis shows that the support for virtual memory for the present generation of MPPs is a challenging task. To support virtual memory efficiently we need to:

(a) reduce the contention for shared resources by increasing the number of I/O nodes and the communication bandwidth.

(b) increase the memory size of each PE, to reduce the page-in and page-out rates of individual node programs.

(c) reduce the service time in case of a page-in by caching pages in the local memory of some other PE or in some data service nodes.

The overall load placed upon the shared resources (communication network and I/O nodes) is determined by the aggregate page fault rate which in turn is determined by the correlation of the paging activities of individual node programs. In our simplified analysis, we have assumed that the intervals at which page faults occur are exponentially distributed. Yet in practice, the SPMD paradigm discussed in §2.3, is often used. In this case, one could expect that different PEs generate page faults at about the same time, that page faults occur in bursts, and that the page fault service time increases dramatically.

The method used by traditional operating systems to hide the page fault latency, namely context switching at the time when a page fault leading to a page-in request occurs, is based upon the fact that the service time for a page fault is considerably larger than the time for context switching. Can the same approach be applied in case of MPPs? To answer this question, we need to examine briefly scheduling on MPPs. In all but a very few applications the node programs of an application need to communicate among themselves. To do so, they need to be active at the same time on different PEs. The scheduling strategy in which all node programs are activated at the same time, then suspended at the same time, activated again at the same time and so on, is called *gang scheduling or co-scheduling*, [1]. Obviously, gang scheduling can be used to hide the latency of page faults if and only if different node programs experience page faults at the same time. Therefore, we need to study the correlation of the paging activity of individual node programs.

To characterize the dynamics of paging for a sequential program we define the *page fault profile* as the number of page faults as function of time. In a uniprocessing environment, the page fault profile of a process can be measured by having a cumulative counter of page faults and by sampling it periodically. In a parallel system we are interested in the dynamics of the total load placed upon the communication and I/O subsystems. To study this dynamics, we define a "cumulative paging profile of a parallel program", by composing the individual paging profiles of individual node programs for all the relevant paging activities like faults, copy-on-write, page-ins, and page-outs.

2.2 A model of parallel program execution.

In this section we are concerned with parallel programs for DMIMD systems. Such a parallel program consists of a set of tasks running concurrently, one task per node or possibly multiple tasks per node in case of multiprocessing nodes. Each task can be either active or suspended. Each active task can be in one of three possible states

(a) *Compute.* The task executes its own code and issues system calls other than I/O and communication ones.

(b) *I/O.* The task has invoked an I/O system call and is waiting for its completion.

(c) *Communication.* The task has invoked a communication system call and is waiting for its completion.

If we want to investigate a certain property of a parallel program for example, its paging behavior, a possible alternative, is to study the dynamics of a set of parameters pertinent to that property. For example, in Section 3.4 we present the parameters relevant to the paging activity of a parallel program under Mach. A first objective is to isolate those parameters which are independent from one another and form a minimum set.

The microscopic behavior of a parallel program consisting of N tasks, π_k, $k = 1, N$ each task going through a sequence of m_k states $S_{j,k}$ with $j = 1$, m_k, will be characterized by the average value $\lambda_{k,j}^{q_i}$ for each of the n parameters q_i with $i = 1, n$ relevant to the property of interest. For each parameter, for each task, and for each state we have a tuple ($t_{k,j-1}$, $t_{k,j}$, $\lambda_{k,j}^{q_i}$) with

$t_{k,j-1}$ – the time where task π_k performed its $(j-1)$ state transition, entering state $S_{j,k}$.

$t_{k,j}$ – the time where task π_k exited the state $S_{j,k}$.

$\lambda_{k,j}^{q_i}$ – the average rate of change of the global counter q_i given by

$$\lambda_{k,j}^{q_i} = \frac{q_i(t_{k,j}) - q_i(t_{k,j-1})}{t_{k,j} - t_{k,j-1}}$$

Several observations are in order. (a) The average rate of change of the parameter q_i is a good approximation of the temporal behavior of the task only if the lifetime of the corresponding state is short. (b) The model is amenable to performance monitoring. One could automatically detect the transition from one state to another, record the values of the parameters in the minimum set every time a state transition occurs, determine the lifetime of the state and compute the average rates. (c) This microscopic characterization is very costly in terms of the amount of information stored. Assume that a parallel program has 1,000 tasks, it runs for 10,000 seconds and has an event rate of 1000 event/second/task. Then the total amount of information recorded during the monitoring of such a program is 120 Gbytes ($12 \times 1000 \times 10,000 \times 1000$, the constant 12 results assuming that each of the three floating point numbers in a tuple needs 4 byte of space). (d) In some cases it will be difficult, if possible at all, to correlate events occurring different tasks. If different PEs have unsynchronized clocks, it is next to impossible to perform such a correlation.

The discrete time model introduced in this section can be used to reduce significantly the amount of data necessary to characterize the behavior of a sequential or parallel program by filtering the raw data obtained through monitoring. For example, assume that parameter q_i has the behavior illustrated in Figure 2a. Figure 2b shows the discrete event representation of η_{q_i} supported by our model. If we accept that values of η_{q_i} lower than a given threshold say $q_{threshold}$ can be neglected, then we can approximate η_{q_i} by a number of peaks raising above a background of level $q_{threshold}$. For example, when we want to model the load imposed upon paging devices, it seems reasonable to filter out the background and retain only the peaks of the page fault profile.

2.3 Modeling the paging behavior of SPMD programs

A commonly used paradigm for solving problems which require considerable amount of computing time using parallel systems, is to partition the data in some manner and to run the same program in all the PEs assigned to the user, each PE executing the same program, but with different data, therefore the name SPMD. This execution mode is compatible with distributed and shared memory MIMD architectures. In SPMD mode, the sequence of instructions executed by different PEs are different due to data dependencies. Often, a special form of data dependency, the dependency of the identity of the PE makes different PEs have a very different flow of control and allows the implementation of a worker-coordinator programming model. In this extended SPMD mode (ESPMD), the coordinator performs functions which are strictly sequential like reading the problem description, computing some initial values, and distributing them to all the workers, then at the end of the computation, collecting statistical information from the workers. A typical code sequence in this mode looks like

```
if (iam == coordinator)
      send initialization_data
else
      receive initialization_data
endif
```

Another extension of the SPMD paradigm is the execution of several cooperating programs, concurrently each program running in a set of nodes. For example, if a computation requires data in a format different from the input provided to it, one could divide the set of PEs into two groups, those in the first group perform the data conversion while those in the second group perform the actual computation on data produced by the PEs in the first group.

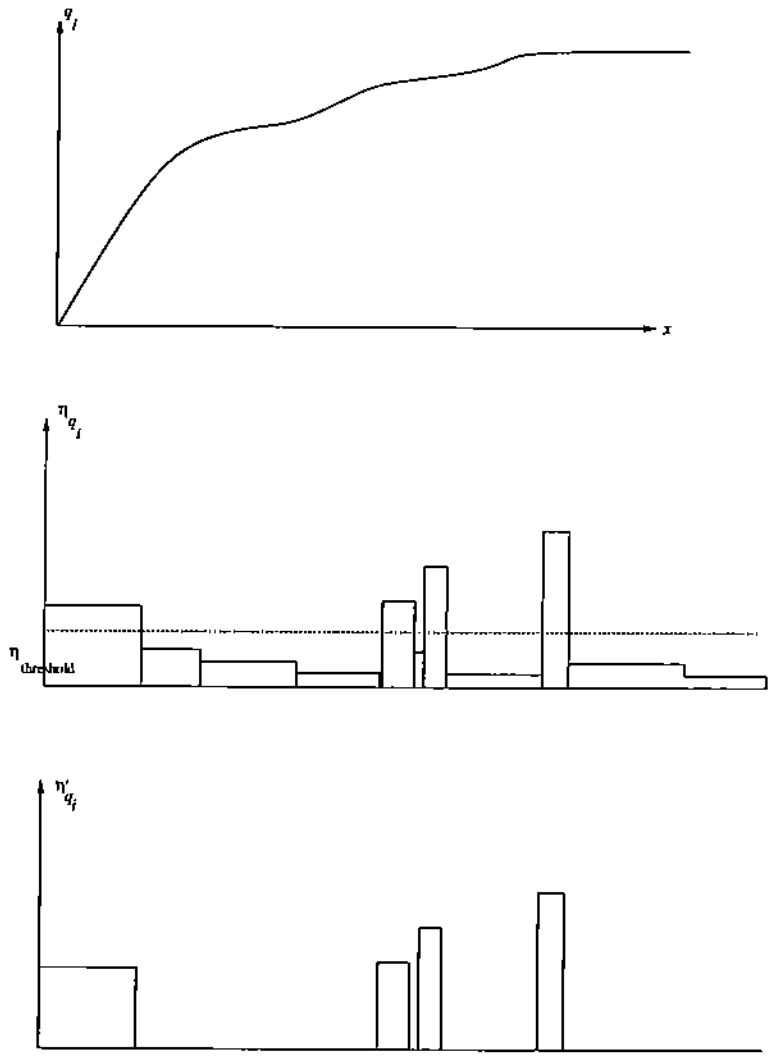


Figure 2: (a)The evolution of the cumulative counter q_i.
(b) The discrete time behavior of q_i. The rate of q_i, η_{q_i} is plotted as function of time. $\eta_{threshold}$ indicates the threshold used to separate the peaks from the background.
(c) The filtering out of the background and retention of the peaks of η_{q_i}.

In the pure SPMD mode, one could reasonably expect similar, but not identical paging behavior for different PEs, while in the ESPMD mode this behavior could be quite dissimilar.

3 Virtual Memory Management in Mach

Mach's virtual memory management consists of machine independent and machine dependent parts. The machine dependent portion is a simple validate/invalidate interface which maintains the hardware address maps. The machine independent portion provides support for logical address maps, memory ranges within this map, and the interface to the backing storage from these ranges via the external memory management interface [2, 13, 14].

The Mach virtual memory system differs from the traditional UNIX virtual memory system by allowing the user to create a *pager* (*external memory manager*) to control the use of memory within portions of a process's address space.

A key element in the design of Mach is the integration of interprocess communication with virtual memory [17]. Memory management techniques, such as copy-on-write and copy-on-reference, are employed to transfer efficiently large amount of data through message passing from a program to another and for efficient network communication. Furthermore, the virtual memory is implemented by mapping process addresses onto *memory objects*, which are represented as communication channels and accessed via messages

In the following sections we describe the implementation of virtual memory management in Mach. In particular, we examine the key Mach memory management operations and how Mach memory objects can be managed outside the kernel by user-level programs.

3.1 Memory Object Abstraction

Memory objects are abstract objects representing collections of data bytes (generally files, pipes, or other data container) that are mapped into virtual memory for reading and writing. Logically, a memory object is a contiguous repository for data, indexed by byte, upon which various operations can be performed. Conceptually, a memory object represents some form of secondary storage.

Unlike other Mach objects, memory objects are not provided solely by the Mach kernel, but can be created and serviced by a user-level data management (*external memory management*) task, or by historical reason, called *external pager*. The data manager is entirely responsible for the contents of a memory object and its permanent storage if necessary.

Like all other objects in Mach system [3, 15], each memory object will have a port associated with it, and may be manipulated by having messages sent to its port. On a page fault, the kernel sends a message to the backing storage port of a memory object to get the data contained in the faulted page.

3.2 Virtual Memory Management

Each Mach task has its own *virtual address space* that all threads within the task fully share. An address space of a task consists of an ordered collection of valid memory regions. The responsibility of providing the virtual memory functionality is shared by both the Mach kernel and the external memory managers. Operations such as *map* and *unmap* of a memory object into a tasks address space are provided by the external memory manger. Yet the Mach kernel provides standard virtual memory functionality including the allocation, deallocation, and copy of virtual memory.

To allocate a new region of virtual memory, the kernel allocates a memory object and uses the *default pager* [6, 13] to manage the object. The *default pager* manages backing storage for memory objects created by the kernel in any of several ways: explicit allocation by user tasks *vm_allocate*, shadow memory objects[13], and temporary memory objects for data being paged out. Physical memory is not allocated until pages in this region are accessed. Copy-on-write sharing is used to perform virtual memory copying efficiently both during task creation (read/write sharing of memory through inheritance) and during message transfer.

The primary role of the kernel in virtual memory management is to manage physical memory as a cache of the contents of memory objects; this makes memory object data available to tasks in the form of physical memory. A remote procedure call requesting data is made by the kernel on the memory object when a page fault occurs and the kernel does not currently have a valid cached resident page. Again, a remote procedure call requesting data flushing is made when the cache is full and the cached page chosen by the kernel to be replaced was modified while it was in physical memory.

3.3 External Memory Management

A pager assumes responsibility for the *mechanics* of page-in and page-out operations over a memory region; Mach kernel retains control over *paging policy*. The presence of external pager is not mandatory. Mach kernel has its own internal pagers to handle ordinary page-in and page-out requests which is the *default* when external pagers are not provided by user-level programs.

The advantage of providing user-level pager for a memory region is allowing the user to define the semantics that apply to that memory object.

3.3.1 Data Structures

Four basic data structures are used within Mach kernel to implement the external memory management interface:

- Task virtual memory *address map* - A task *address map* is a directory mapping each of many valid address ranges to a memory object, the offset within that object along with protection and inheritance information.
- *Virtual memory object* - Each *memory object* used in an address map is represented by an internal *memory object* structure. Information kept in this structure includes: the port used to access the memory object, its size, the number of address map references to the object, and whether the kernel is permitted to cache the memory object when no address map references remain.
- *Resident pages structure* - Each *resident page* structure corresponds to a page of physical memory; a page belongs to just one *object*. The resident page structure records the memory object and offset into the object, along with the access permitted to that page by the data manager. Reference and modification information provided by the hardware is also saved here.
- *Page replacement queues* - Page replacement uses several *page-out queues* linked through the resident page structures (physical memory). Three queues are built: the *active queue* contains all pages currently in use (mapped) in LRU order, the *inactive queue* is used to hold pages being prepared for page-out (reference bit cleared), and the *free queue* keeps pages not holding any data.

3.3.2 Fault Handling

The Mach kernel fault handler is invoked when the hardware tries to reference a page for which there is no valid mapping or for which there is a protection violation. Figure 3 illustrates the actions taken during a page fault handling [17].

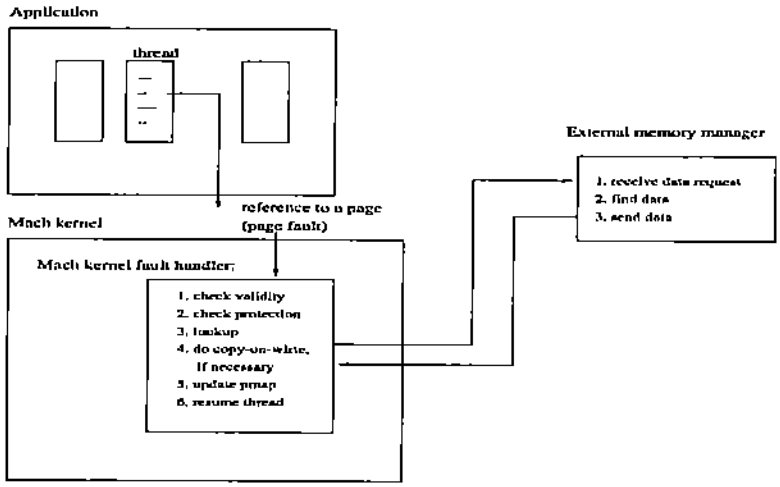


Figure 3: Page fault handling: Mach kernel fault handler

3.4 OSF/1 Virtual Memory Statistics

The OSF/1 Mach 3 [9] maintains statistics on the use of virtual memory since the time the kernel was booted for the processor on which that kernel is executing in the *vm_statistic* structure. The pertinent information for the paging activity given in *vm_statistic* is listed below.

- *free count*, *active count*, *inactive count* - report the status (size) of *page-out queues* at a given moment.
- *wired count* - gives the number of pages "forced" to be and stay resident (cannot be paged out) in a given moment.
- *zero-fill count* - gives the number of pages allocated and initialized with zero (usually a page is zero-filled when it is allocated with *vm_allocate* using the *default* memory manager.)
- *page-ins* - gives the number of requests for pages from the kernel to the pager.
- *page-outs* - gives the number of pages that have been paged out.
- *faults* - gives the number of times page faults have been detected.
- *cow-faults* - gives the number of copy-on-write faults (deferred evaluation optimization) have occurred.

- *lookups, hits* - control the number of memory cache lookups and hits.

The data in the *vm_statistic* structure consists of *non-cumulative* and *cumulative counters*. Non-cumulative counters give the status of the paging queues when the *vm_statistic* is read. For example, *active count* reports the number of pages currently in use (in the active queue), *free count* the number of pages not holding any data, and so on. Cumulative counters keep the number of occurrences of an activity from the kernel was booted. For example, *faults* reports the number of page faults occurred since the boot time to the moment when the *vm_statistic* struct is read. *Zero-fill count, page-ins, page-outs, faults, cow-faults, lookups, hits* are cumulative counters. *Free count, active count, inactive count,* and *wired count* are non-cumulative counters.

The type of statistic plays an important role in the form of processing its value. From a sequence of statistics collected from the system in different intervals, it makes sense to take the weighted (by interval) mean value of non-cumulative counters and take the weighted rate value of cumulative counters.

The average rate of a certain indicator of the paging activity, e.g., page faults, page-ins, etc. is computed by taking the difference between two consecutive readings divided by the time elpased between the two readings.

For our research purpose, the important measures of the paging activity are: *faults, page-ins, page-outs,* and *cow-faults.* Based on the behavior of these counters we are able to characterize the paging activity of the application under study. Other counters, such as *free, active, inactive* count, *lookup,* and *hits,* are mostly related to the status of the physical memory cache, which depend primary on the kernel.

4 Parallel Program Profiling

Following the parallel program execution model described in the previous section (see §2.3), we investigate the paging behavior of a parallel program by observing the dynamics of the paging parameters generated by the program.

An event-driven *parallel profiling library* is provided to monitor and profile the execution of the parallel programs on the Paragon^{TM} XP/S System running the OSF/1 Mach Operating System. Following the Mach terminology we call a node program a *task.* During the execution of each task snapshots of paging statistics at each state transition are collected. The set of parameters related to paging activities in the OSF/1 Mach kernel is found in the *vm_statistic* structure, collected at the time a state transition occurs.

The steps for profiling a parallel program are:

(a) Instrumentation of the source code. A pre-processor detects the points of state transition, such as I/O and communication system calls, and substitutes the corresponding system call by the counterpart profiler routine. Then the profile library, either the version for C or for Fortran programs, is linked to the application's source code to generate "profile-able" application execution code. This transformation of the source code is fully automated.

(b) Parallel program profiling. During the profiling phase, each task of the parallel program generates trace records which are stored in a separate trace file. The trace records are collected according to the dynamics of the task execution path.

Each trace record contains a time stamp which marks the occurrence time of the corresponding event. The Paragon^{TM} XP/S System provides a global clock which is 1 microsecond accurate across all nodes (the "Reprogrammable Performance Monitoring counters"), allowing the reconstruction of the global event sequence.

(c) Post-processing trace data analysis. The trace data collected during the profiling phase are further processed to extract relevant information about the parallel program paging activities. Several tools, described in the subsequent section, are provided to analyze those data.

4.1 Parallel Profiling Library

The design of the parallel profiling library strives to guarantee accuracy of the collected data by minimizing the intrusion of the profiling library (see also §6.2) in the parallel program.

The following steps are taken to ensure limited intrusion:

- A compact format for the trace record. A minimum set of parameters relevant to the paging activities are collected in addition to the execution state and the time stamp.
- Data collection frequency is limited to events of interest, such as task state transition.
- I/O operation frequency, such as writing trace records into trace files, is reduced by using buffered I/O routines.
- Off-line data analysis is adopted to avoid additional computation during the program profiling phase.

Trace data are collected at events of interest, in this case the transition of states generated by I/O or communication system calls. To profile a parallel application, I/O and

communication system calls are substituted by their counterpart profiler routines. A profiler routine generates two trace records in addition to the system call itself. Thus, during the execution of a profiler routine, one trace record is collected before executing the system call and another right after its completion.

The following code illustrates the functioning of the profiling library.

Original Source Code	*Profiled Source Code*	*Execution State*
. . . .	. . . .	
a = 1234;	a = 1234;	
b = my_computation(a);	b = my_computation(a);	Compute
x[i] = b;	x[i] = b;	
csend(O_data, o_array, o_size);	**Ps_csend**(O_data, o_array, o_size);	Communication
a = a * arcsin(x[i]);	a = a * arcsin(x[i]);	
b = my_computation(b);	b = my_computation(b);	Compute
x[i] = b;	x[i] = b;	
crecv(I_data, i_array, i_size);	**Ps_crecv**(I_data, i_array, i_size);	Communication
. . . .	. . . .	Compute

A trace record is a tuple (t_j, S_j, V_j) with

a) t_j – the time when the record for the event j is generated.

b) S_j –the type of the event j. The event types are: *START_IO, END_IO, START_MSG,* and *END_MSG.*

c) V_j – the set of values of global counters, q^i with $i = 1, n$, at time t_j.

In the example illustrated above, a task executing *Ps_csend()* generates two consecutive trace records: $(t_{j-1}, START\_MSG, V_{j-1})$ and (t_j, END_MSG, V_j). These two trace records provide the following information:

1. A *compute* state ended at time t_{j-1}.
2. A *communication* state started at time t_{j-1} (START_MSG) and ended at time t_j (END_MSG). The average rate of change of the parameters is calculated based on values recorded for V_{j-1} and V_j.
3. A *compute* state started at time t_j. This *compute* state ends when a new trace record is issued, i.e. when the following communication or I/O event occurs.

Since the trace records for each task of the parallel program are stored sequentially in its corresponding trace file, two consecutive trace records in the trace file of task k provide n tuples $(t_{k,j-1}, t_{k,j}, \lambda_{k,j}^{q_i})$ with $i = 1, n$ (one for each parameter), in addition to the corresponding value of the execution state $S_{j,k}$.

4.2 Profiling Library User Interface

The profiling library consists of profiler routines for I/O and communication system calls, and a few routines providing the user with profiling flow control. The user can turn on and off the profiling, may generate additional trace records using the *Ps_profile_trace* call and the other event control primitives in Table 1.

One of the problems in the event-driven profiling approach is the variable duration of a state. Since the trace records are collected at points of state switching, a long state is equally described as a short state is. It is possible that an application stays in a state (e.g. *compute*) for a long period of time or that an application presents a very small number of state switches. In those applications a very small number of trace records are generated for a long period of time.

The small amount of trace records describing a long period may not reflect accurately the paging behavior of the application under examination. To remediate it, the library routine *Ps_profile_trace()* is provided to increase the number of trace records collected during the *compute* state. The user may insert *Ps_profile_trace()* calls in the source code, and at execution time each call to *Ps_profile_trace()* generates a trace record containing the snapshots of paging parameters. Because of the intrusion and overhead to the program execution, this routine should be used only it is necessary; the indiscriminate use of it can alter the behavior of the program.

Table 1 summarizes the profiling library user interface.

5 Data Reduction and Peak Selection

In this section we study the correlation of the paging activities of individual node programs in the SPMD execution mode and introduce a methodology for reduction, interpretation, and analysis of the paging activities called "skyline analysis".

Communication			*I/O*		*Event Control*
Msg Passing		Global Ops			
Ps_csend	Ps_crecv	Ps_gsync	Ps_cread	Ps_open	Ps_init
Ps_csendrecv		Ps_gdsum	Ps_cwrite	Ps_close	Ps_printstat
Ps_isend	Ps_irecv	Ps_gisum	Ps_iread	Ps_stat	Ps_sleep
Ps_isendrecv		Ps_gssum	Ps_iwrite	Ps_lseek	
Ps_msgdone	Ps_msgwait		Ps_iodone	Ps_eseek	Ps_profile_label
Ps_hsend	Ps_hrecv		Ps_iowait		Ps_profile_trace
Ps_hsendx	Ps_hrecvx				
Ps_hsendrecv	Ps_gsendx				

Table 1: The Profiling Library User Interface

5.1 Data Reduction

For most of the parallel applications studied, the profiles for different paging activity indicators show transients of high paging activity intermixed with longer period of low activity. This raw paging activity profile can be examined using the visualization tools described in the third part of this paper.

The objectives of data reduction are (a) to expedite the identification of paging activity bursts and observe the system response under heavy paging work load, and (b) to reduce the amount of data needed to describe the parallel program paging behavior.

The data reduction is composed by the following three elements: peak selection, background filtering, and skyline representation.

1. *Peak Selection*

   During the execution of a SPMD parallel program, there are usually several bursts of paging activity (called "*peaks*") at certain periods intermixed with other low activity periods, called "*background*".

   In the pure SPMD mode, peaks could happen approximately at the same time for all nodes, in contrast with the ESPMD mode, where the coordinator may present a paging profile different than that of other nodes. In both cases, the high load imposed to the system in a given moment, when accounting for paging requests from all nodes, is a key element for characterizing the paging activity of a SPMD program.

   To compose the paging profile of parameter q_i for a node k, we select the N_k largest peaks occurred during the execution in node k. The number N_k may vary for each

node according to the following selection criterion (see §5.2)

$$N_k = \min(\text{all peaks within the selection range}, N_{max})$$

The *selection range* is upper bounded by the maximum value across all nodes of the rate of parameter q_i, η_{q_i}, and lower bounded by a *background cut-off* value $\eta_{threshold}$. N_{max} is the maximum number of selected peaks.

After selecting N_k peaks, we further reduce the amount of data needed by analyzing the amplitude of the parameter in the neighbor event records for each selected peak. Records with similar peak amplitude are "aggregated" into one peak with a duration equal to the duration of all aggregated peaks.

2. *Background Filtering*

   The selected peaks are sparse in time; the interval between two successive peaks is populated with a number of trace records with small amplitudes, called a *background region.* The trace records in a background regions are reduced to one "background record" $B_j : (t_s, t_e, \lambda_{B_j}^{q_i})$ representing the interval of the background and the weighted average value of the parameter q_i of all records previously within the background region.

$$\lambda_{B_j}^{q_i} = \frac{\sum_i (\Delta t_i \lambda_i)}{\sum_i \Delta t_i}$$

   with Δt_i being the interval of a reduced record and λ_i its corresponding parameter value.

3. *Skyline Representation*

   The paging profile is reduced to a much more compact discrete time representation composed by peaks and background regions, each one represented by a tuple $(t_s, t_e, \lambda^{q_i})$ as described in §2.3.

   For example, for $N_k = 15$ selected peaks, we represent the paging profile of parameter q_i of node k by using at most $2N_k - 1 = 31$ tuples instead of the large number of trace records collected during the program profiling.

   The graphical representation of the model is a "*skyline*" representing the overall paging behavior of a node.

The data processing methodology presented above has been shown effective in reducing the amount of data needed to describe the paging behavior without losing relevant information. The advantage of the small set of data include: (1) a compact discrete time model and (2) ease the correlation and analysis of data across nodes.

Several graphic visualization tools are provided to expedite the understanding of the paging behavior, such as the time line of a parameter collected in the trace records before filtering – *raw trace data*, the time line of the compact paging profile of the same parameter – *skyline representation*, along with other graphic tools used during the peak selection process and the data analysis after the filtering process.

Some comments are in order. (1) The maximum number of peaks to be selected N_{max} and the background cut-off limit B_{q_i} are parameters which can be tuned to a particular application. The cut-off limit varies for different parameters and for different applications. (2) The amount of the space needed for a parameter q_i of node k in a parallel program has an upper bound of $(2N_{max} - 1)$ records (or tuples). The effectiveness of the space saving depends on the dynamic of the application, i.e., the number of records gathered during the profiling phase.

5.2 Peak Selection Criterion

To examine the paging parameter q_i based on the collected trace data, the peak selection procedure uses two counterbalanced parameters: the constant value N_{max} and the background cut-off value $\eta_{threshold}$.

Depending on the purpose of the data processing, more weight is given to one or other parameter. A small N_{max} reduces significantly the amount of trace data because only the few higher peaks (N_{max}) are selected, and all others are neglected and merged with the background regions. This indiscriminate data filtering process creates a paging behavior description which is not accurate with the profiled program's behavior.

Since our primary purpose is not space saving but to provide a simplified but accurate description of the profiled program's behavior, the *background cut-off level* is the most important parameter in the selection criterion. For the resulting skyline representation to be as close as possible to the real program's behavior, we must define an adequate cut-off level.

Two graphs, one showing the total number of selected peaks, and the other the number of peaks selected in each node, both for different cut-off levels, are provided to help defining an adequate background cut-off value for the purpose of trace data analysis. A compact data set (one with fewer peaks) facilitates the first rough analysis, but a detailed data set

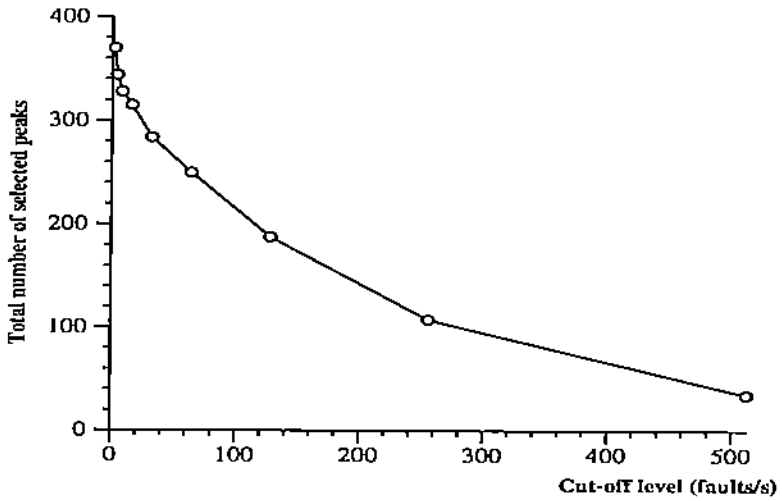


Figure 4: The relationship of the total number of peaks to the cutoff level for the fault rate. The parallel program Envelope is running in 16 nodes on a PARAGON XP/S system.

describes more accurately the program behavior.

Figure 4 shows an example of the total number of selected peaks for the page fault rate, for a SPMD program using different cut-off levels. As expected, the smaller is the cut-off value, the larger is the number of selected peaks.

Similarly, Figure 5 shows the number of peaks selected in each node for the same program of Figure 4. The number of peaks selected in each node may be slightly different, although the curve of the number of selected peaks across nodes for a given cut-off level maintains the same shape as the cut-off level varies. When the cut-off value increases, the number of selected peaks in each node decreases. The same characteristics of both graphs are observable in all three program execution states.

The cut-off value for the peak selection is chosen by the user depending upon the level of detail needed and the parameter under examination, using graphs like those in Figures 4 and 5. Such graphs are provided by our analysis package.

6 Case Studies

In this section we discuss the applications we have monitored, and present and analyze the data we have collected. The conclusions drawn from our analysis of a few programs have to

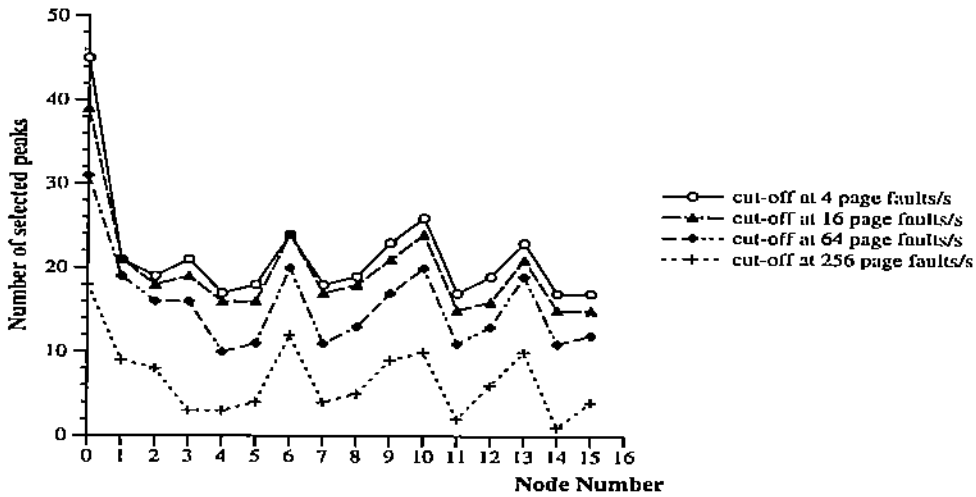


Figure 5: The number of peaks selected per node for different cutoff rates. The parallel program Envelope is running in 16 nodes on a PARAGON XP/S system.

be confirmed by additional data before they can be used to improve the design of massively parallel systems and the mechanisms for resource sharing in such systems.

6.1 The applications

We have concentrated our attention on a few applications in the area of computational biology we helped develop over the past few years [4, 11]. The programs we have studied are used for the 3-D atomic structure determination of large macromolecules like viruses. These programs are:

(a) The Envelope program used for real space electron density averaging.

(b) The FFTsynth, a program used for transformation from reciprocal to real space by means of 3-D FFT.

(c) The Recip program used to correlate calculated structure factors with observed ones.

A brief outline of the computations and the algorithms used by these programs follows.

6.1.a The Envelope program

The input for this program is a 3-D lattice with $nx \times xy \times nz$ points. Every grid point with coordinates (x_0, y_0, z_0), has an electron density ρ_{x_0,y_0,z_0}. Symmetry operators: π_1, $\pi_2, \ldots, \pi_n$ allow us to associate to every grid point (x_0, y_0, z_0) n other points, (x_1, y_1, z_1), $(x_2, y_2, z_2) \ldots (x_n, y_n, z_n)$, related to it by non-crystallographic symmetry. The electron density at every grid point is replaced by the average value of the electron density of all the points related by non-crystallographic symmetry.

$$\rho_{x_0,y_0,z_0} = \frac{\sum_{i=0}^{n} \rho_{x_i,y_i,z_i}}{n+1}.$$

The parallel algorithm used for averaging is based on a partition of the 3-D lattice into small volumes (bricks). Each PE is assigned a number of bricks to transform, but it needs access to the entire data space (the entire lattice) because points related by the non-crystallographic symmetry are scattered throughout the lattice. The program implements a shared virtual memory and operates in two modes, (1) the disk mode (DFS), and the data caching in the nodes mode (DAN). The memory maps of the two modes are presented in Figure 6.

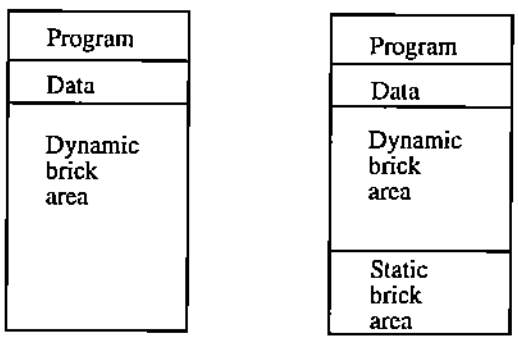


Figure 6: The memory layout for the Envelope program. In the DFS mode (6a) each processor fetches bricks on demand from the disk file into the dynamic brick area. In the DAN mode (6b) the bricks are cached in the static brick area of all nodes. When a processor needs a brick not available locally, it uses interrupt driven communication to fetch it from another processor's static brick area.

6.1.b The FFTsynth program

The program performs 3-D Fourier synthesis. The parallel algorithm consists of three steps. In step 1, the 3-D volume is partitioned into xy-slabs, groups of consecutive xy-planes. A 1-D FFT in the y direction is carried out. In step 2, a global transposition takes place and each PE ends up with an xz-slab, a collection of xz-planes. In the third step, each PE carries out 2-D FFTs on all xz-planes assigned to it.

6.1.c The Recip program

The inputs to the program are: (1) A very large sorted file containing several millions of records of observed data. Each record consists of the coordinates in the reciprocal space (the Miller indices), the amplitude, and additional information relative to the reflection. (2) A file of unsorted data containing calculated structure factors. The first step of the computation is to sort (2). Then each record of (2) is compared against (1). If a match of the Miller indices is found, then the calculated amplitude is replaced by the observed one from (1).

6.2 The environment

The measurements were performed on a Paragon XP/S system with 66 compute nodes, 2 I/O nodes and 3 service nodes with 32 Mbyte of memory per node. The system is currently running Paragon OSF/1, Release 1.0.4, Patch R1.1.6.

We discuss briefly the intrusion due to our monitoring [12]. Table 2 shows the size of the original load module and of the load module of the instrumented programs. As we can see, the instrumented code is 5-11% larger than the original code. Table 3 presents the effects of instrumentation upon the execution time for the FFTsynth program. The instrumented code takes about 15% more time.

	Original code	Instrumented code
Envelope	606	641
FFTsynth	601	622
Recip	417	461

Table 2. The size of the load module (in KBytes) for the original and the instrumented code.

# of PEs	Original code	Instrumented code
8	167	181
16	147	167
32	118	136

Table 3. The execution time (in seconds) of the original and the instrumented code for the FFTsynth program.

Yet another form of intrusion which affects the data obtained through monitoring, is due to interactions of the program being monitored with programs running concurrently in other partitions of the system. This type of interactions is due to contention for the I/O and communication resources, and always leads to an increase of the execution time. Even when a program is not instrumented, its execution time may vary by a significant amount for the same input data depending upon the activities of its competitors. In our monitoring process, the correlation of the peaks of activity in time is affected by this type of intrusion.

While the first type of intrusion, the one due to our measurements *cannot* be eliminated, there are costly ways to eliminate the second type by using the machine in an exclusive fashion.

6.3 Event and state information

Our methodology for monitoring the paging activity of a parallel program is based upon detecting transitions from one state of the task to another and recording the paging activity data collected by the kernel at time of the transition. A parallel program can be in one of the following states: compute, I/O and communication.

In this section we present synthetic data intimately related to our model, namely

- the total number of events recorded during the execution of the program,
- the average number of events per node,
- the total time spent by a program in each state, and
- the average duration of an event

for all the programs we have monitored. Figures 7,8,9 and 10 present these data for the Envelope program in the DFS and DAN mode, the FFTsynth program and the Recip program.

The data we collect to study the paging activity of parallel programs is extremely useful to understand the behavior of a parallel program, the way it uses the resources of the parallel system and to determine means for improving the performance of the parallel program.

For example, Figures 7a and 7b show that the Envelope program in DFS mode performs a large number of I/O operations and there are virtually no communication events. Figure 7c shows that the I/O bandwidth of the system is insufficient. As a result there is no gain in using 64 PEs, the execution time with 64 PEs is essentially the same as the one with 32 PEs, about 1,300 seconds. While in case of 16 nodes, the time spent in the I/O state is less than 10% of the total execution time (200 seconds for a 2200 seconds execution time), it represents more than 60% when 64 PEs are used (about 1,000 seconds out of 1,300 seconds total execution time). Figure 7d provides additional arguments that the I/O is the bottleneck, the average duration of an I/O event increases from about 0.08 seconds for 16 PEs to about 0.2 seconds for 32 PEs and about 0.5 seconds for the 64 PEs.

This analysis justifies our approach used in the DAN execution mode of the Envelope program, to cache data across nodes. Figures 8a, 8b, 8c and 8d present this mode of execution using 16 PEs in two different experiments called A and B, and 32 PEs. As expected, the number of I/O events is considerably lower, but there are many more communication events compared with the DFS mode. The total execution time is reduced, about 1,600 seconds in 16 (versus 2,200 seconds) with 16 PEs, and about 800 seconds (versus 1,300 seconds) with 32 PEs. Figure 8c also shows that the total time spent in the compute state is a fairly large fraction of the total time (more than 90% for the 32 PE case) and that the fraction of time spent in the I/O state is insignificant. Figure 8d illustrates that the execution time of a program is influenced by the other program running concurrently in other partitions of the system. In case of experiment A, the average duration of an I/O event is larger than that of experiment B, because in case of A, other programs requesting I/O were running concurrently.

Figures 9a-9d describe the data collected for the FFTsynth program. There are two options to perform the global exchange (see §6.1b), the *in-place* and the *external* global exchange. In the first case, each PE sends and receives data from every other PE and no I/O is involved. In the second case, each PE writes its intermediate results to a file and then when all PEs have finished writing their data, all read. The first mode was used in the 8, 16 and 32 PEs experiments, the second mode in the 64 PEs experiment. The high level of contention for I/O is revealed by Figures 9c and 9d for the second mode of operation and is reflected in the very poor performance of this mode. The execution time with 64 PEs is about 140 seconds, while the one with 16 PEs is 95 seconds and the one with 32 PEs is 50 seconds. Figure 9b shows that the number of communication events per node increases linearly with the number of nodes, due to the in-place global exchange algorithm.

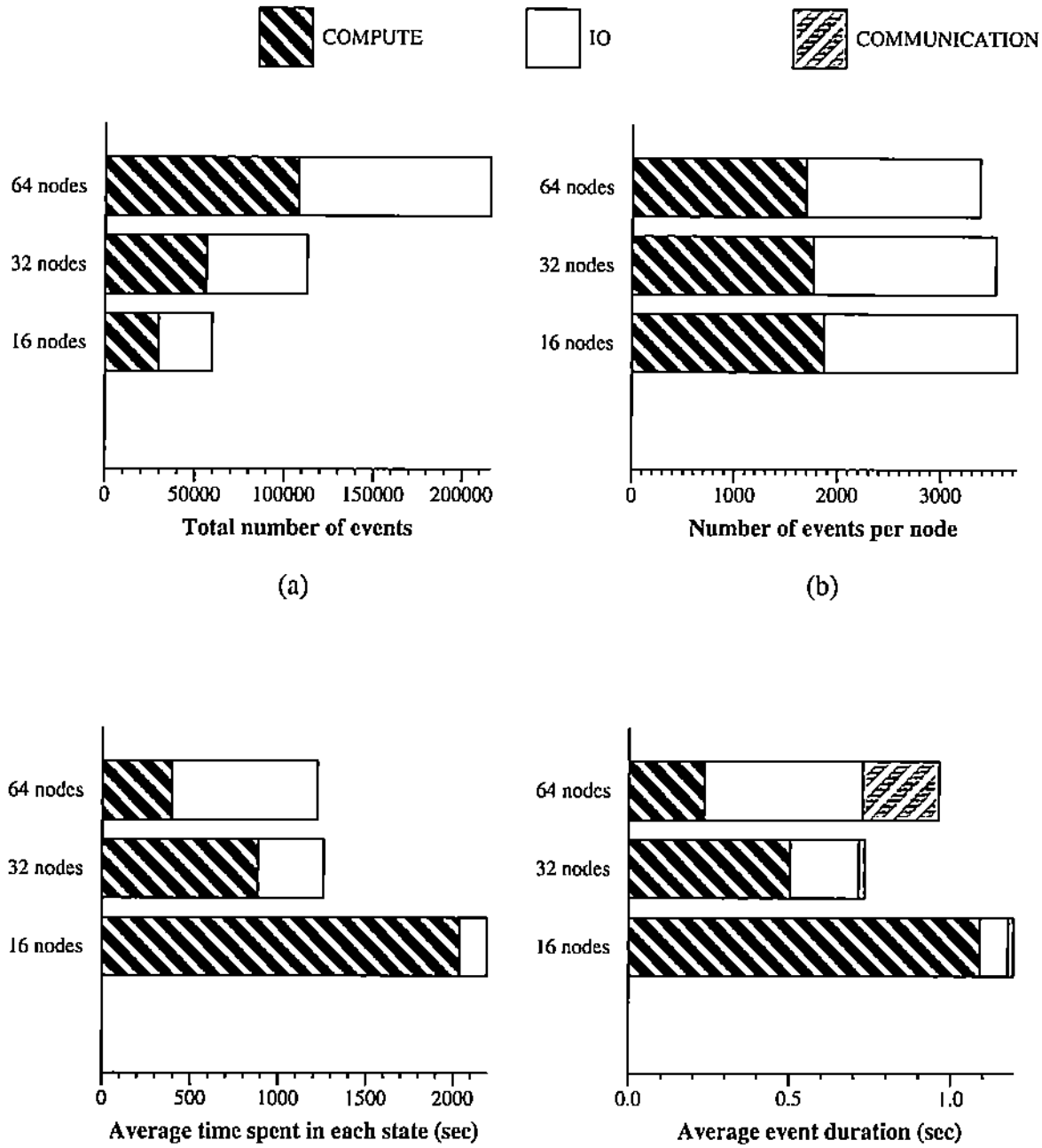


Figure 7: Summary of information concerning the number of events, the average lifetime of a state and the average lifetime of an event for the Envelope program running in DFS mode.

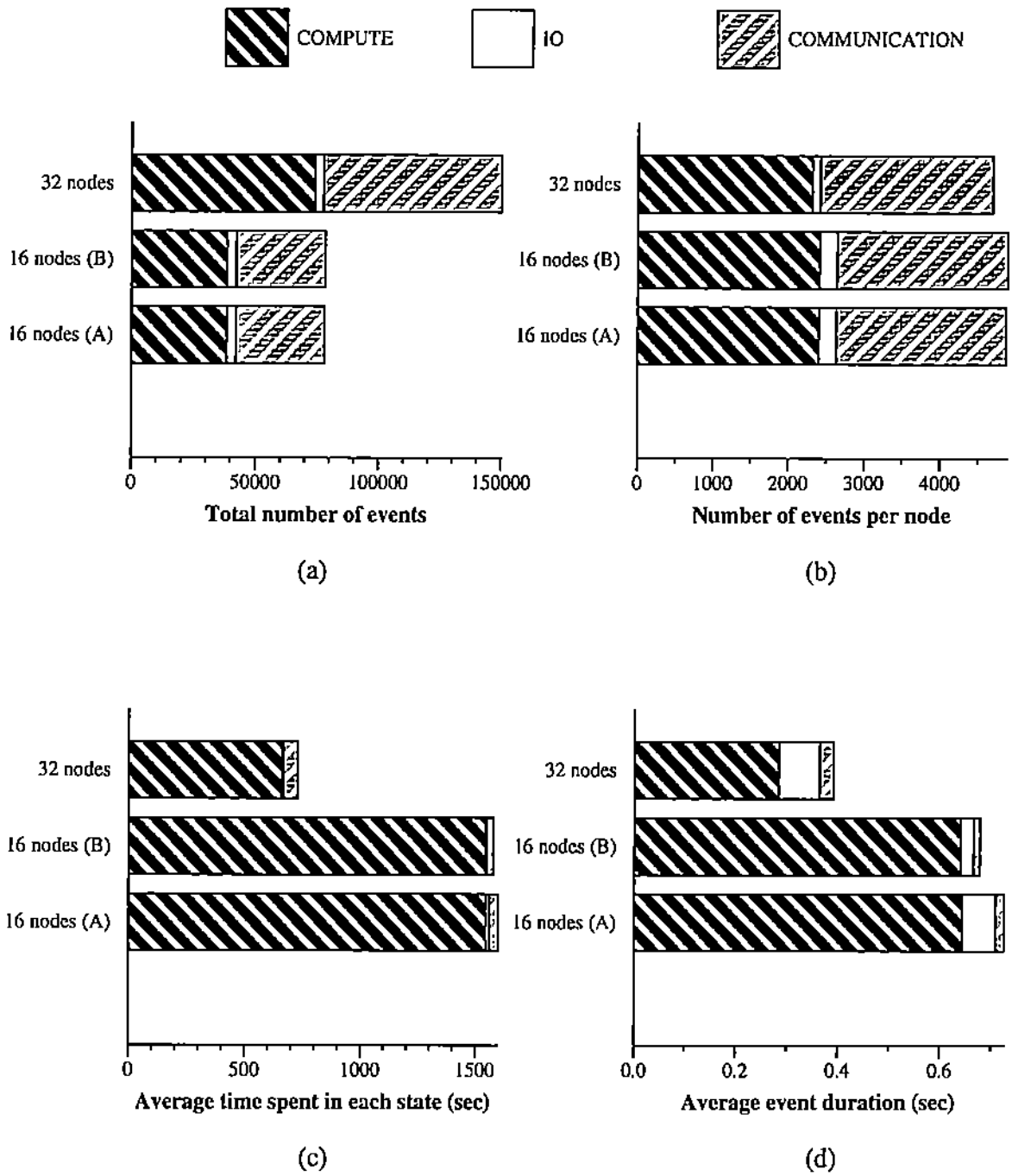


Figure 8: Summary of information concerning the number of events, the average lifetime of a state and the average lifetime of an event for the Envelope program running in DAN mode.

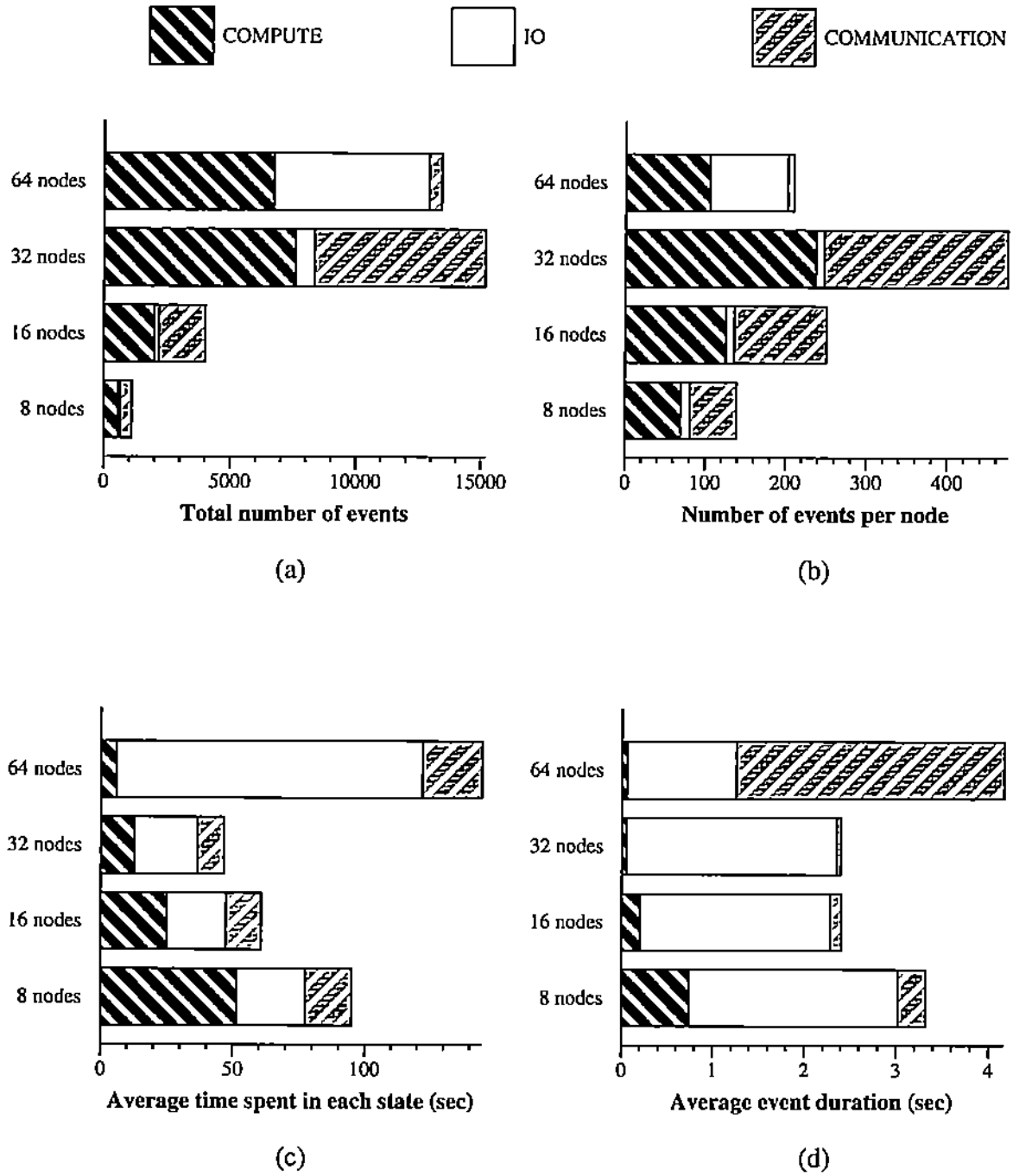


Figure 9: Summary of information concerning the number of events, the average lifetime of a state and the average lifetime of an event for the FFTSynth program.

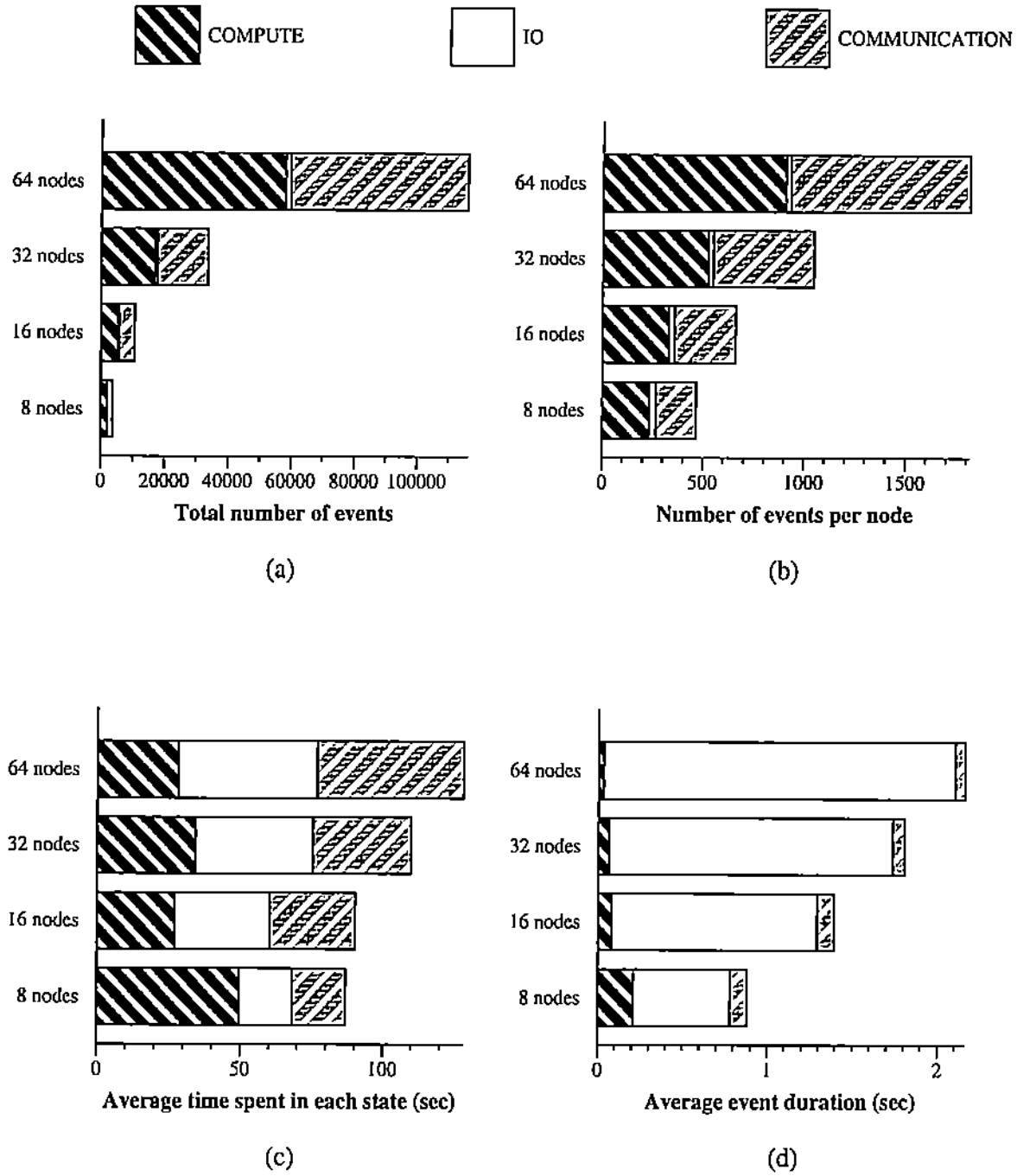


Figure 10: Summary of information concerning the number of events, the average lifetime of a state and the average lifetime of an event for the Recip program.

Figures 10a-10d present the data for the Recip program. We observe that again the number of communication events per node increases linearly with the number of nodes as expected, due to the sorting algorithm which requires a global exchange. Though there are few I/O events, the time spent in the I/O state with 16 or more PEs is about 1/3 of the total execution time, due to contention for I/O illustrated by the substantial average duration of an I/O event, Figure 10d.

6.4 The Amplitude and Time Correlation of the Paging Activities of the Individual Node Programs

The methodology used to study the correlation of the paging activities of individual PEs is to isolate the peaks of activity and to study how their amplitude and time of occurrence relate to each other. For example, Figure 11 shows the page fault data obtained for the Envelope program running in 16 nodes in DFS mode. Figure 11a, 11b and 11c show the amplitude correlation in the three states, compute, I/O and communication, while Figure 11d, 11e and 11f show the same data for time correlation. From Figure 11a, we see that in the compute state, we have isolated 16 peaks. Peaks 1, 2 and 5 exhibit the most dissimilar behavior. For example, among the 16 PEs, the lowest rate of page fault for the first peak of activity is slightly lower than 250 faults/sec and the highest rate observed is slightly lower than 950 page faults/second. Figure 11d shows that the first seven peaks of the page-fault activity occurred within the first 10-15 seconds and the time elapsed between the first and the last occurrence of a certain peak is less than 50 seconds. We see that the time elapsed between the first and the last occurrence of a certain peak increases as the peak id increases. For example, the 15-th peak occurred first after about 220 seconds and the last PE experienced this peak some 170 seconds later. The data presented in Figures 11 to 58 is organized as follows:

- Figures 11 to 22 present the data for Envelope in the DFS mode on 16, 32 and 64 PEs.
- Figures 23 to 30 present the data for Envelope in the DAN mode on 16 and 32 PEs.
- Figures 31 to 42 present the data for the FFTsynth program running in 8, 16 and 32 PEs.
- Figures 43-58 present the data for the Recip program running in 8, 16, 32 and 64 modes.

For each run we present the page fault, page-in, copy-on-write and page-out paging activity indicators. When running the Envelope program in the DFS mode (see Figures 11 to 22), there are very few communication events. For this reason, Figures 11e to 22e show few

peaks of activity (sometime none) and the time correlation graphs do not show any data. The few communication events occur very early during the execution of the program and the are separated by milliseconds rather than tens or hundreds of seconds.

7 Conclusions

An accurate analysis of the paging activity of a parallel program can only be done provided that there is enough hardware and kernel support for monitoring the execution of a program. The minimal hardware support requires the existence of synchronized clocks for all PEs. The kernel should support creation of a trace record for every paging event to identify the time when the event occurs, the type of fault and possibly other information. Such an accurate monitoring activity is very costly and very intrusive.

We have opted for a less accurate, yet less intrusive and less costly approach, based upon sampling of the event counters maintained by the kernel. The sampling is done at the time of a state transition or as an explicit user request. The time spent by a node program in a state varies widely, therefore our sampling of the paging activity counter occurs at irregular intervals and we are forced to use average rates to measure the paging activity.

It is difficult and risky to derive general conclusions from a few experiments and we will stress only those observations which are confirmed by all our measurements and have a plausible explanation. In the absence of other results in the literature, we have decided to include all our measurements to allow the reader to identify other trends we have not observed and to verify our conclusions.

All our measurements carried out for different programs running on a different number of PEs show that *in the paging activity, there is a substantial dissimilarity among the node programs even for SPMD programs.* This is plausible due to data dependencies, but it is still very surprising to see that one PE experiences an average rate of 50 page-in/second and another one close to 300 (Figure 16a, the first peak in the COMPUTE state). Yet such discrepancies are rather common for all parameters. The correlation in time of different peaks of activity, shows the same trend. For programs which use asynchronous algorithms like the Envelope in which each node works independently, the difference in time of occurrence from the first PE to the last PE increases with the peak id.

This observation has a profound impact upon scheduling on MPPs which use gang scheduling of all tasks in a partition to allow them to communicate with one another. The high latency of a page-in request cannot be hidden by context switching as in the case of traditional operating systems. Even moderate rates of page-ins may lead to a substantial increase of the execution time of a parallel program and to fairly poor utilization of resources

of the system, due to communication among tasks.

There are two possible solutions to this problem. The first is to attempt to reduce the number of page-ins by increasing the amount of real memory available or by anticipating the faults and bringing in pages before they are actually needed. There are no general solutions we are aware of for this last approach. The second possible solution of this problem is to reduce the latency of a page-in by having dedicated nodes with a fairly large memory acting as swap devices.

Unfortunately, we are unable to answer an important question related to the paging activity of parallel programs. Given the data characterizing the paging activity of a program running with n PEs we cannot predict the paging activity when running with m PEs.

8 Acknowledgements

The authors want to express their thanks to Dr. Denise Ecklund from the Supercomputer Systems Division of Intel for many insightful discussions which have helped clarify many aspects of our work. Many thanks to Dr. Zhongyun Zhang who has helped us install the parallel programs on the system used for testing.

References

[1] M.J. Atallah, C. Lock, D.C. Marinescu, H.J. Siegel, and T.L. Casavant, "Models and Algorithms for Co-Scheduling Compute-Intensive Tasks on a Network of Workstations," *Journal of Parallel and Distributed Computing*, vol 16, 1992, pp. 319–327.

[2] R.F. Rashid. "Threads of a New System," In: *UNIX Review*,

[3] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming Under Mach*, Addison-Wesley Publishing Co., 1993. 481 pages.

[4] M.A. Cornea-Hasegan, D.C. Marinescu, and Z. Zhang, "Data Management for a Class of Iterative Computations on Distributed Memory MIMD Systems," *Concurrency: Practice and Experience*, vol 6(3), pp. 205–229, 1994.

[5] E.P. George and G.M. Jenkins, "Time Series Analysis: Forecasting and Control," revised edition, Holden-Day, Oakland, California, 1976.

[6] D.B. Golub, R.P. Draves. "Moving the Default Memory Manager out of the Mach Kernel," *Proceedings of the Usenix Mach Symposium*, pp. 177–188, 1991.

[7] R. Hofmann, R. Klar, N. Luttenberger, B. Mohr, and G. Werner, "An Approach to Monitoring and Modeling of Multiprocessor and Multiprocessor Systems," In: *Performance and Distributed and Parallel Systems*, (T. Hasegawa, H. Takagi, Y. Takahashi, eds.), Elsevier Science Publishers B.V. (North-Holland), IFIP, pp. 91–110, 1980.

[8] Intel Corporation, Paragon^{TM} OSF/1 User's Guide, Inter Supercomputer Systems Division, Beaverton, Oregon, 1993.

[9] K. Loepere. "Mach 3 Kernel Interfaces," In: *Open Software Foundation*, Carnegie Mellon University, 1992.

[10] A.D. Malony, D.A. Reed, and D.C. Rudolph, "Integrating Performance, Data Collection, Analysis and Visualization," In: *Performance Instrumentation and Visualization*, (M. Simmons, R. Koskela, eds.), Addison Wesley, 289 pages, 1990.

[11] D.C. Marinescu, J.R. Rice, M.A. Cornea-Hasegan, R.E. Lynch, and M.G. Rossmann, "Macromolecular Electron Density Averaging on Distributed Memory MIMD Systems," In: *Concurrency: Practice and Experience*, vol 5(8), 1993. pp. 635–657.

[12] D.C. Marinescu, J.E. Lumpp, T.L. Casavant, and H.J. Siegel, "Models for Monitoring and Debugging Tools for Parallel and Distributed Software," In: *Journal of Parallel and Distributed Computing*, vol 9, 1990, pp. 171–184.

[13] R.F. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W.J. Bolosky, and J. Chew, "Machine-independent Virtual Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers*. v.37, n.8, pp. 896–908, 1988.

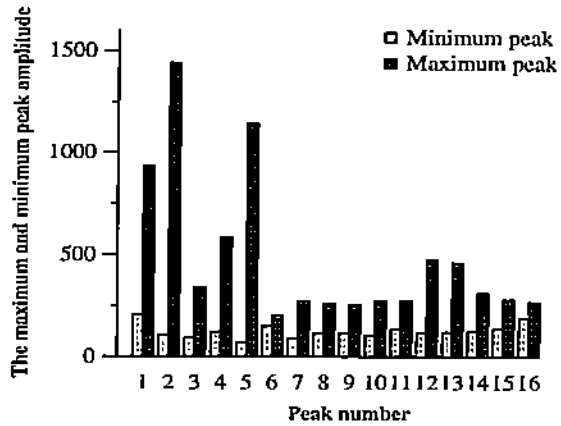
[14] R.F. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi, "Mach: A Foundation for Open Systems," In: *Proceedings of the 2nd Workshop on Workstation Operating Systems (WWOS2)*, IEEE, pp. 109–113, 1989.

[15] A. Silberschatz, J.L. Peterson, and P.B. Galvin, "The Mach Operating System," In: *Operating Systems Concepts*, Third Edition, Chapter 16, pp. 597–628, Addison-Wesley Publishing Co.

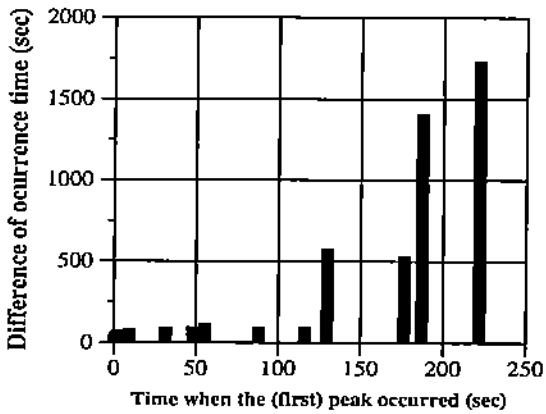
[16] D. Wybranietz and D. Haban, "Monitoring and Measuring Distributed Systems," In: *Performance Instrumentation and Visualization*, (M. Simmons, R. Koskela, eds.), 289 pages, 1990.

[17] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implemen-

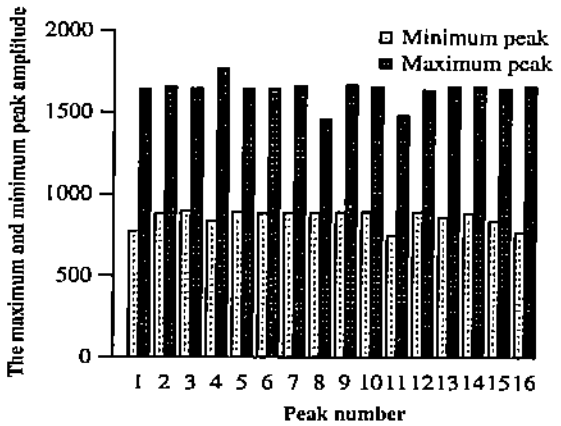
tation of a Multiprocessor Operating System," *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 63–76, 1987.



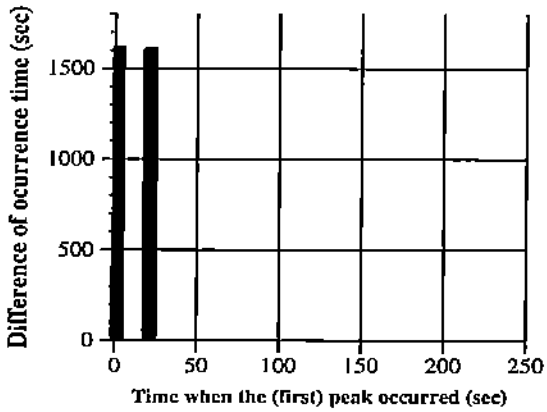
(a) COMPUTE state: amplitude correlation



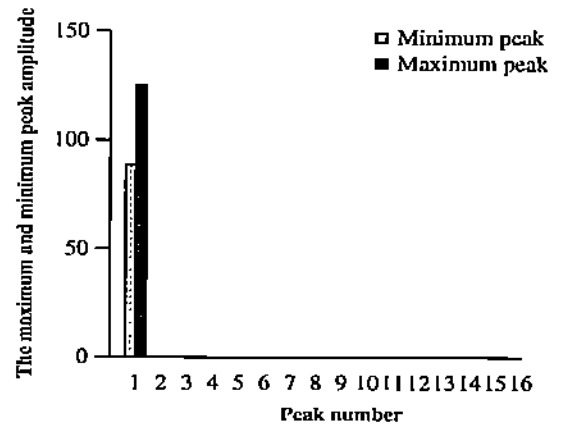
(d) COMPUTE state: time correlation



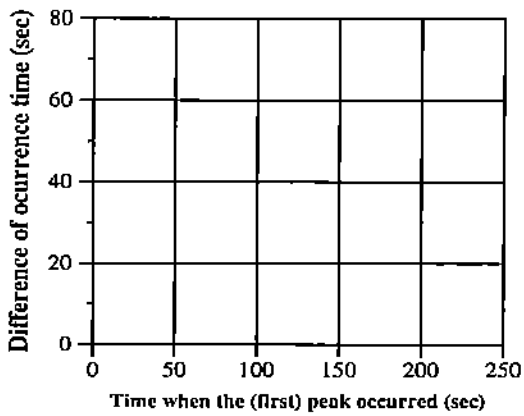
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

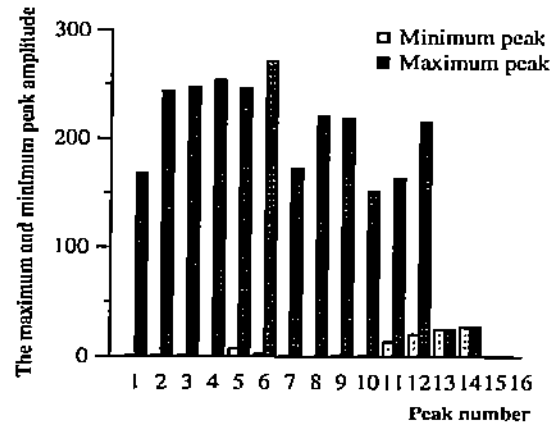


(c) COMMUN. state: amplitude correlation

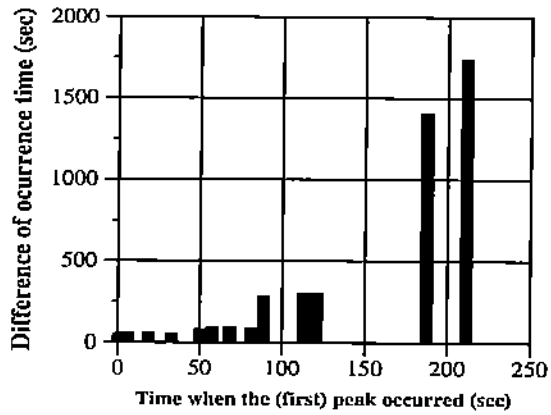


(f) COMMUN. state: time correlation

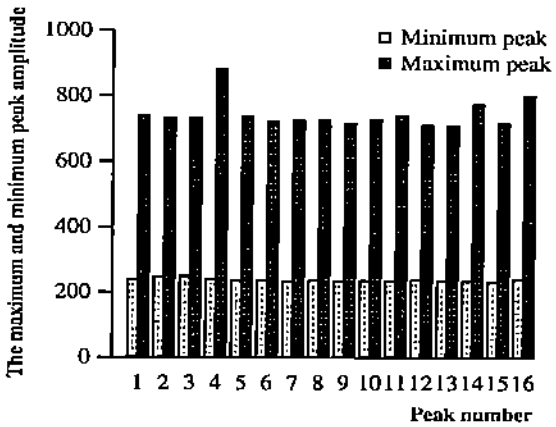
Figure 11: The page fault analysis. The correlation of amplitude and time of occurrence for the peaks of page fault activity for the Envelope (DFS mode) program running in 16 nodes.



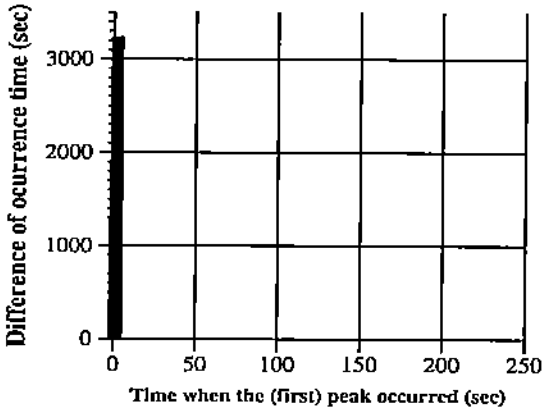
(a) COMPUTE state: amplitude correlation



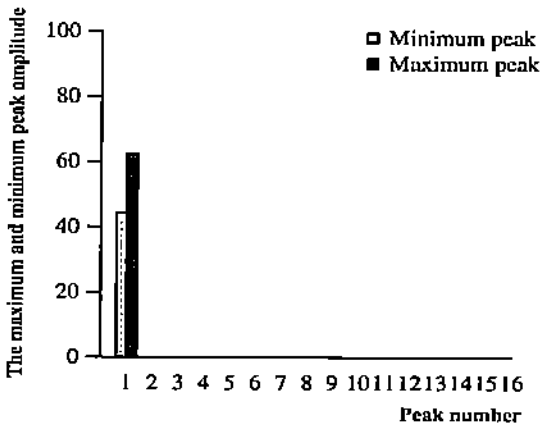
(d) COMPUTE state: time correlation



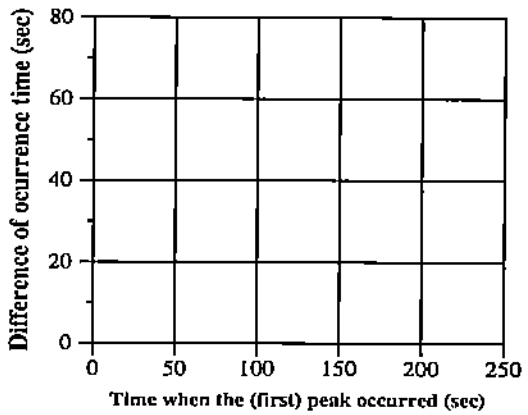
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

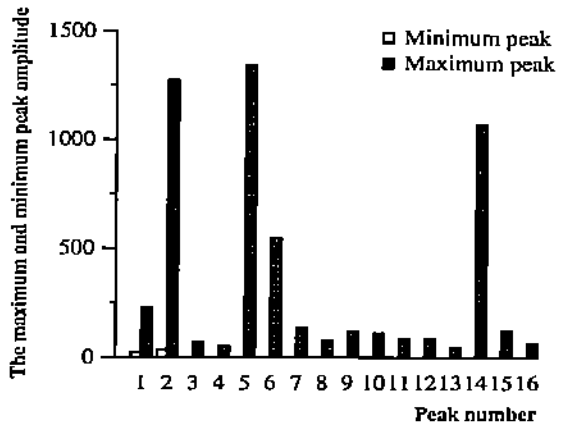


(c) COMMUN. state: amplitude correlation

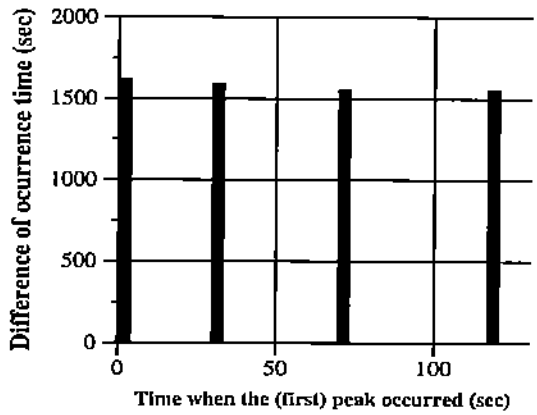


(f) COMMUN. state: time correlation

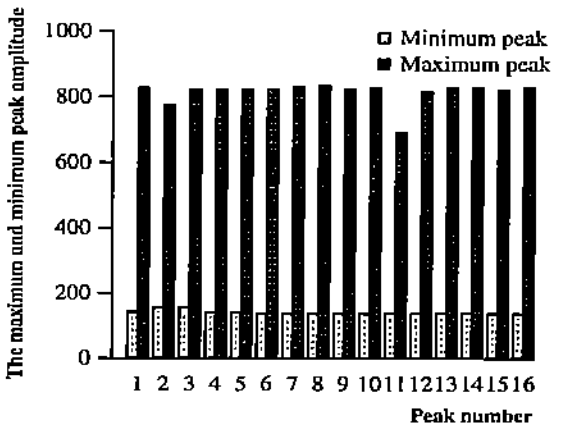
Figure 12: The page-in analysis. The correlation of amplitude and time of occurrence for the peaks of page-in activity for the Envelope (DFS mode) program running in 16 nodes.



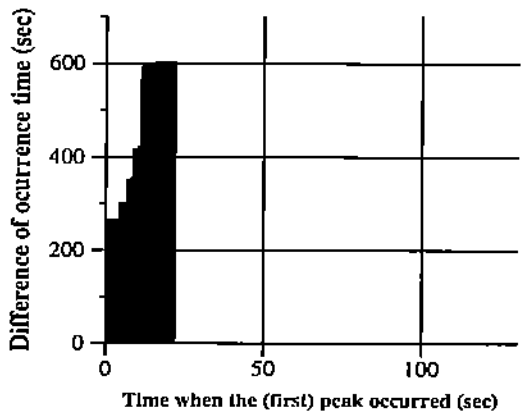
(a) COMPUTE state: amplitude correlation



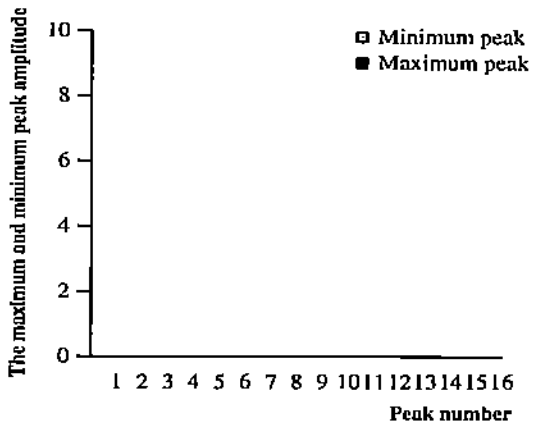
(d) COMPUTE state: time correlation



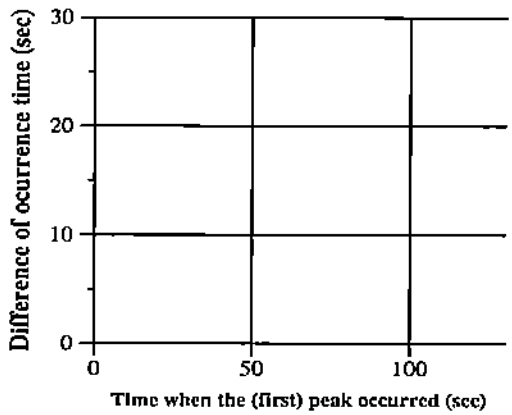
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

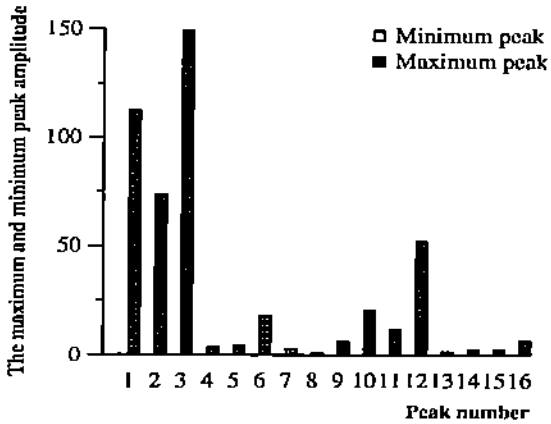


(c) COMMUN. state: amplitude correlation

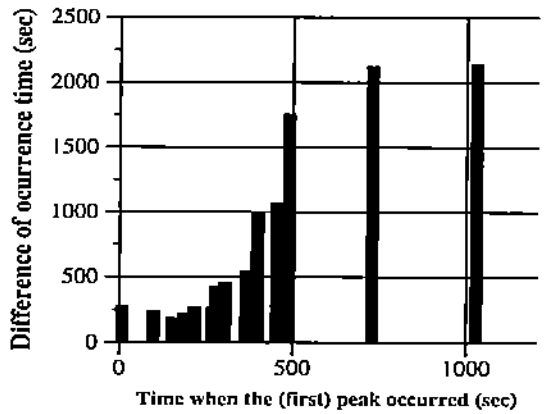


(f) COMMUN. state: time correlation

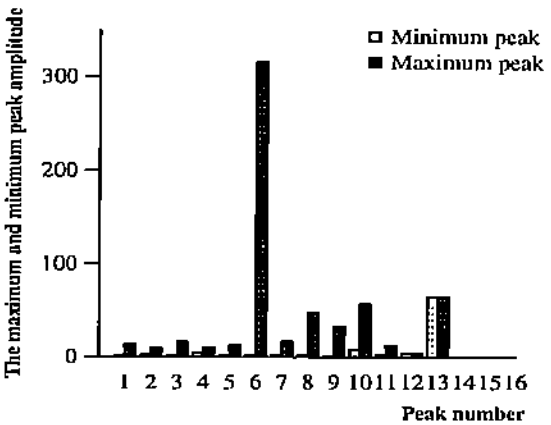
Figure 13: The copy-on-write fault analysis. The correlation of amplitude and time of occurrence for the peaks of cow fault activity for the Envelope (DFS mode) program running in 16 nodes.



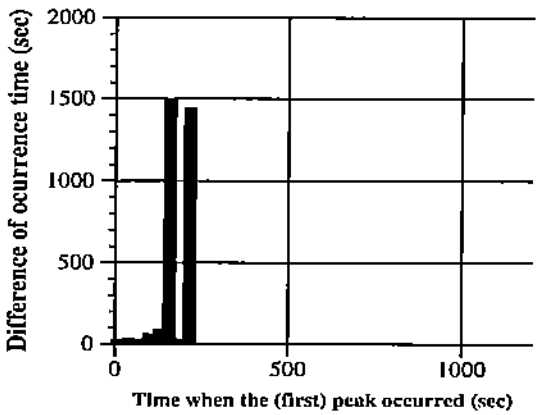
(a) COMPUTE state: amplitude correlation



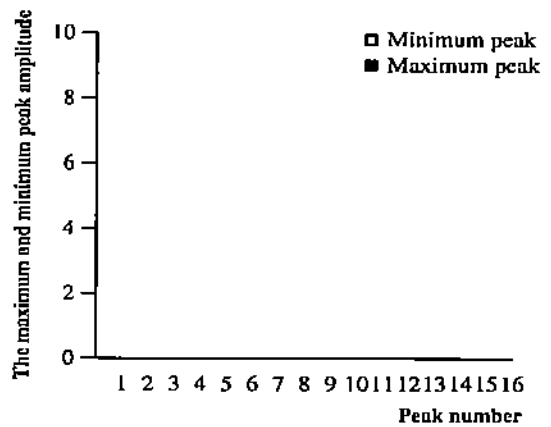
(d) COMPUTE state: time correlation



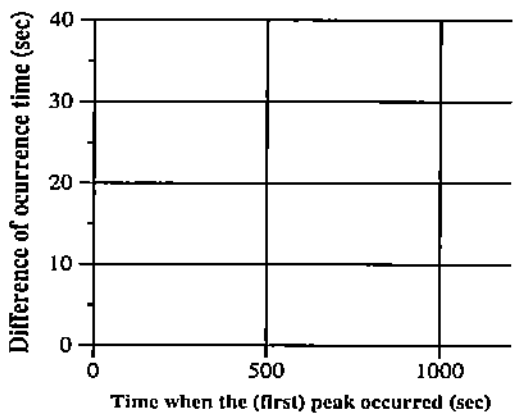
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

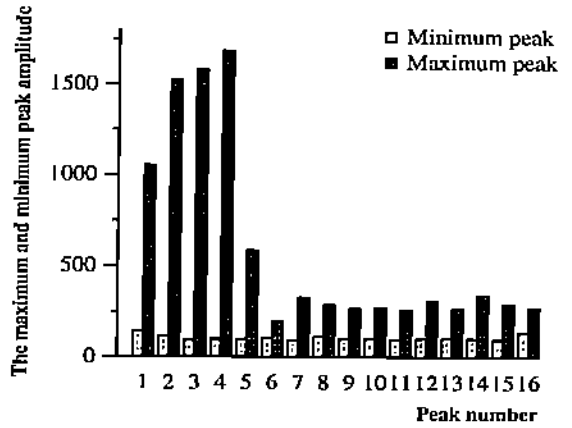


(c) COMMUN. state: amplitude correlation

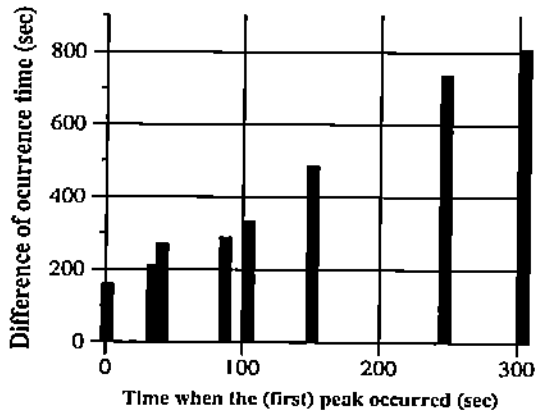


(f) COMMUN. state: time correlation

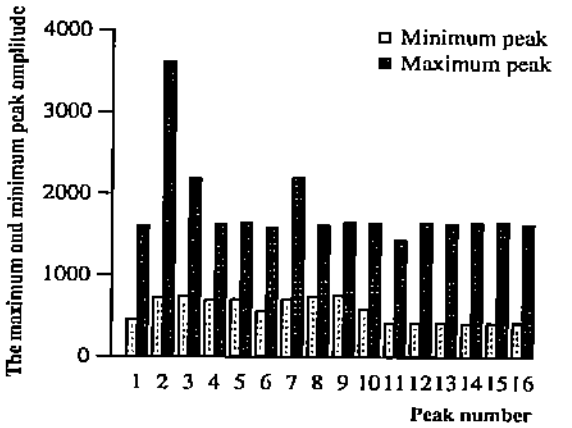
Figure 14: The page-out analysis. The correlation of amplitude and time of occurrence for the peaks of page-out activity for the Envelope (DFS mode) program running in 16 nodes.



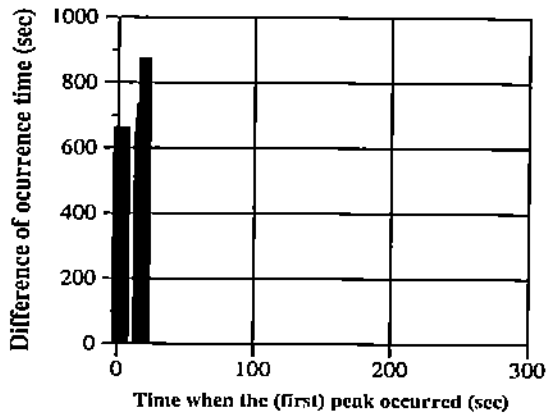
(a) COMPUTE state: amplitude correlation



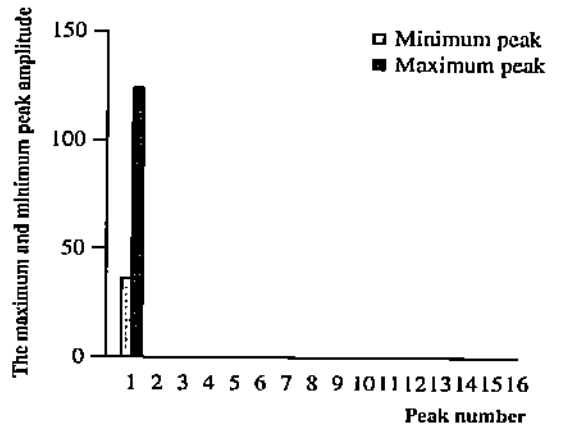
(d) COMPUTE state: time correlation



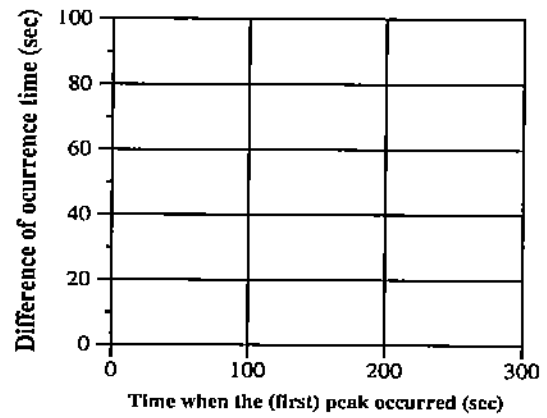
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

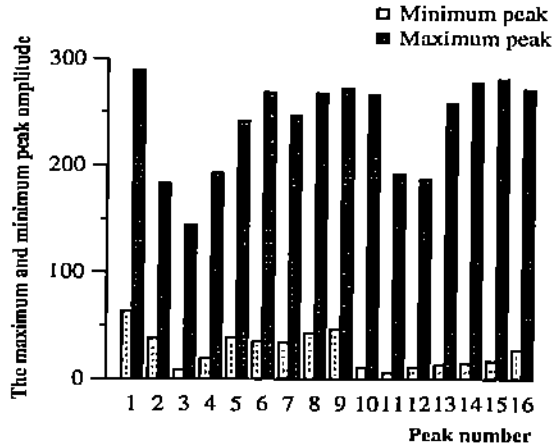


(c) COMMUN. state: amplitude correlation

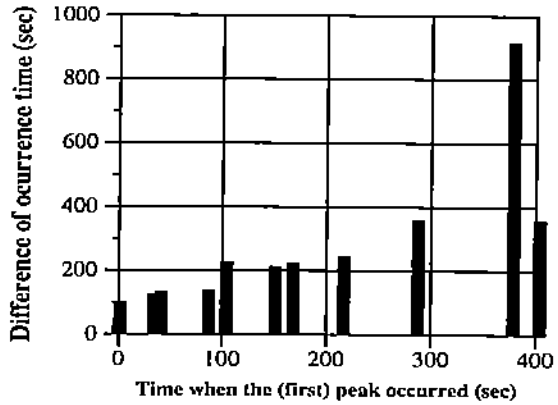


(f) COMMUN. state: time correlation

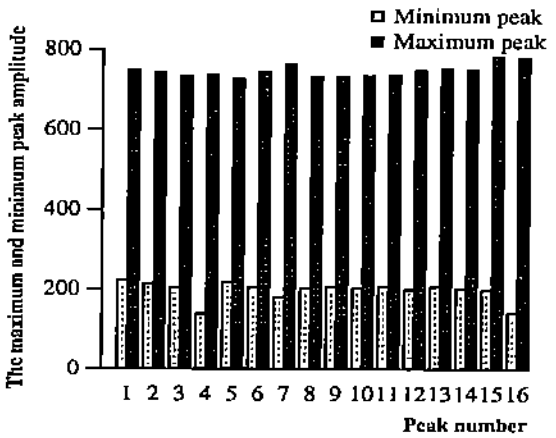
Figure 15: The page fault analysis. The correlation of amplitude and time of occurrence for the peaks of page fault activity for the Envelope (DFS mode) program running in 32 nodes.



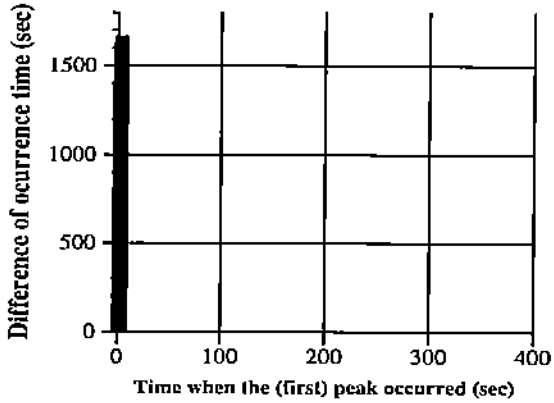
(a) COMPUTE state: amplitude correlation



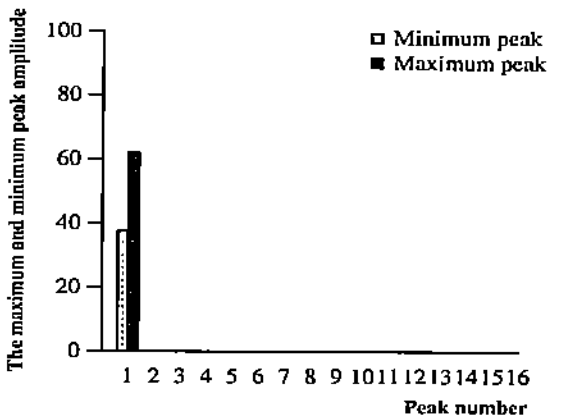
(d) COMPUTE state: time correlation



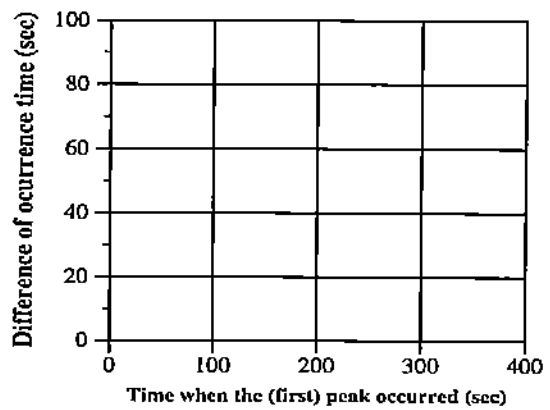
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

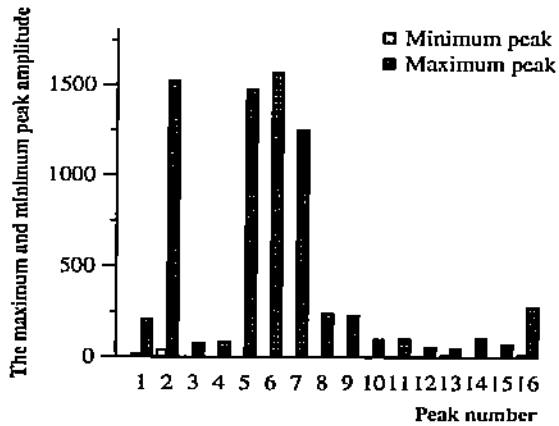


(c) COMMUN. state: amplitude correlation

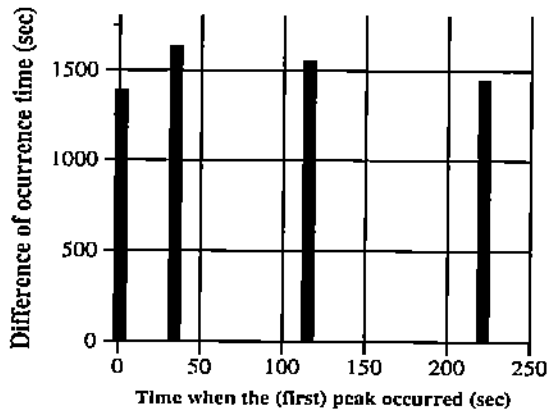


(f) COMMUN. state: time correlation

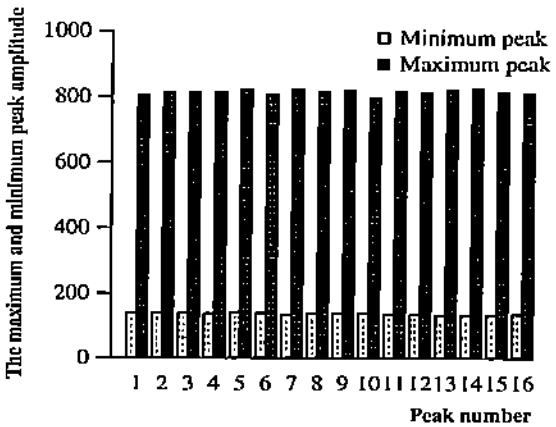
Figure 16: The page-in analysis. The correlation of amplitude and time of occurrence for the peaks of page-in activity for the Envelope (DFS mode) program running in 32 nodes.



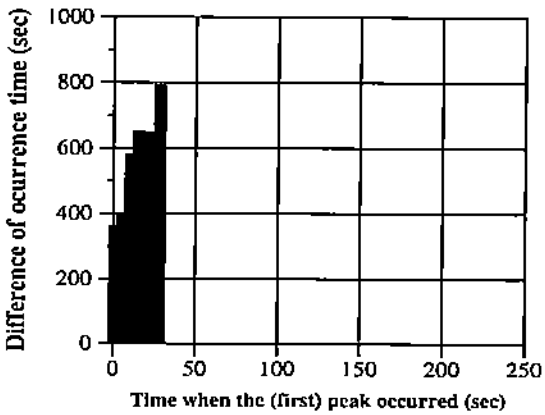
(a) COMPUTE state: amplitude correlation



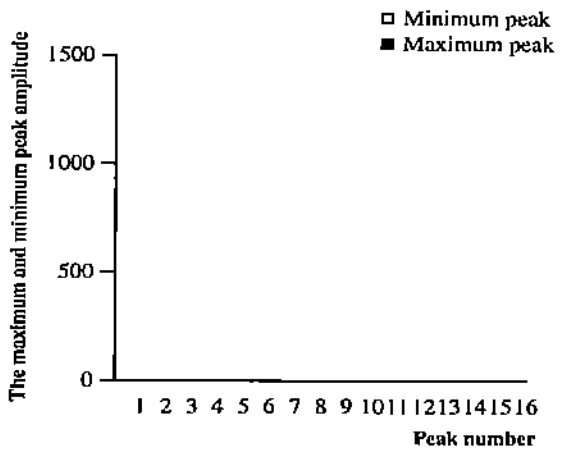
(d) COMPUTE state: time correlation



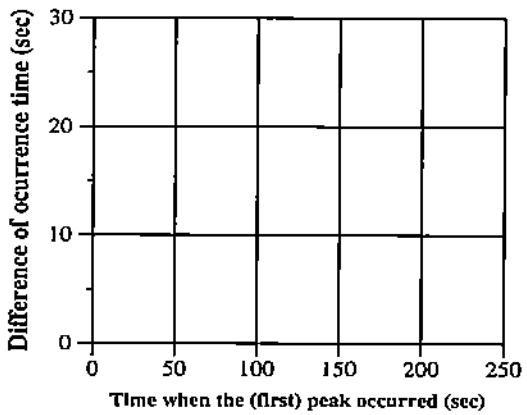
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

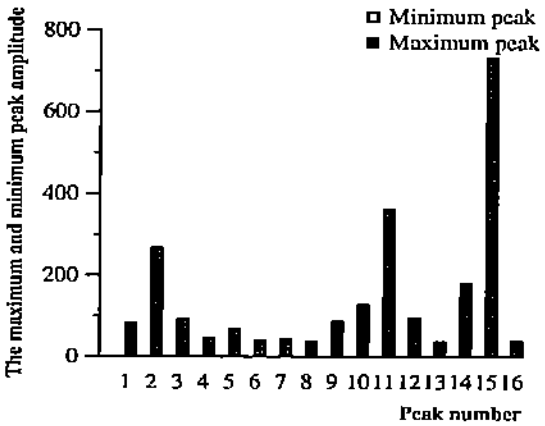


(c) COMMUN. state: amplitude correlation

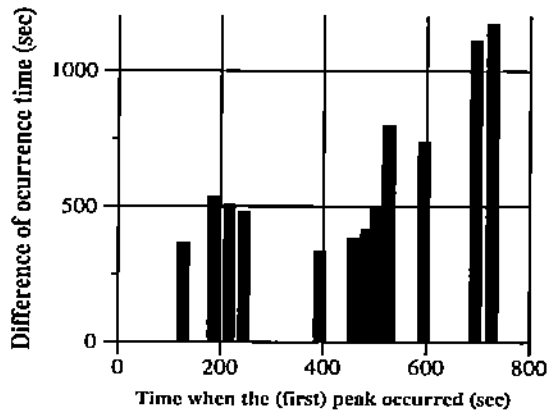


(f) COMMUN. state: time correlation

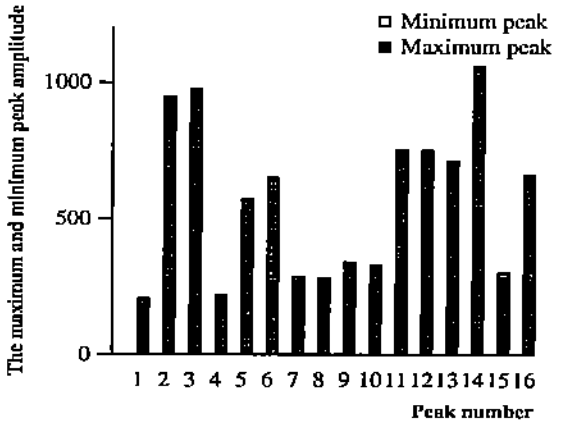
Figure 17: The copy-on-write fault analysis. The correlation of amplitude and time of occurrence for the peaks of cow fault activity for the Envelope (DFS mode) program running in 32 nodes.



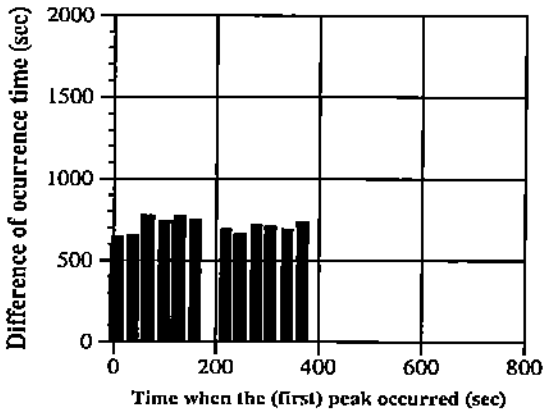
(a) COMPUTE state: amplitude correlation



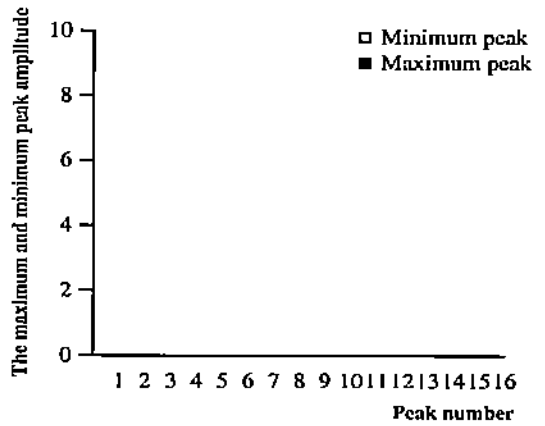
(d) COMPUTE state: time correlation



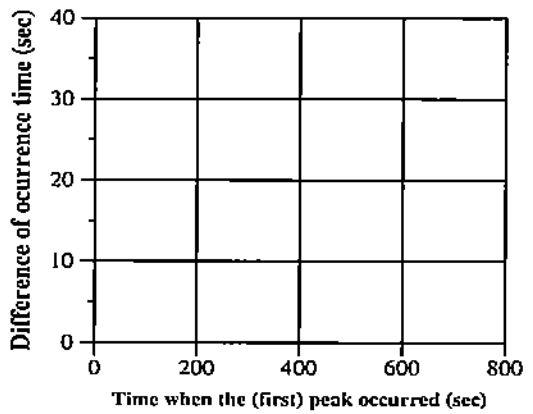
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

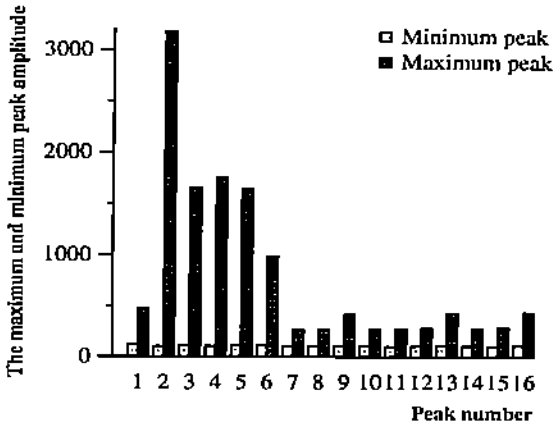


(c) COMMUN. state: amplitude correlation

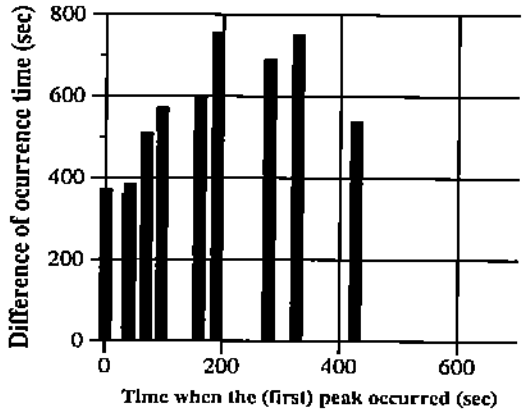


(f) COMMUN. state: time correlation

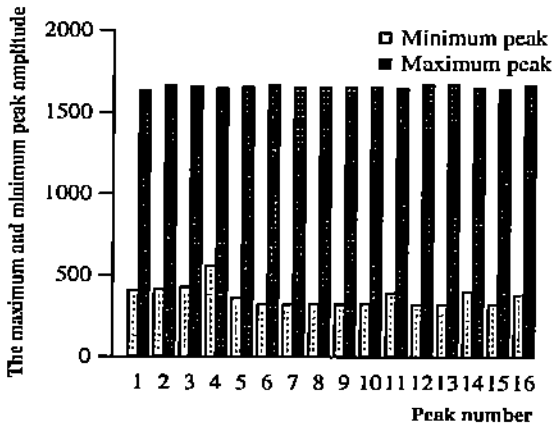
Figure 18: The page-out analysis. The correlation of amplitude and time of occurrence for the peaks of page-out activity for the Envelope (DFS mode) program running in 32 nodes.



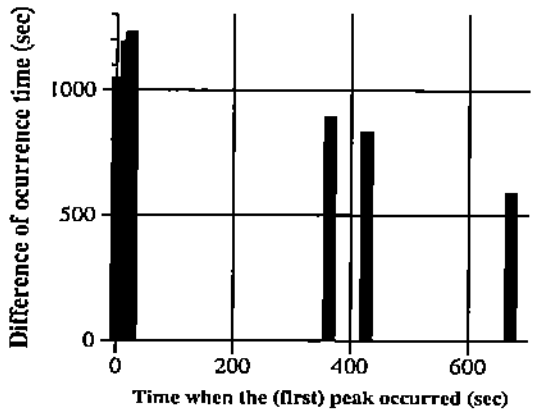
(a) COMPUTE state: amplitude correlation



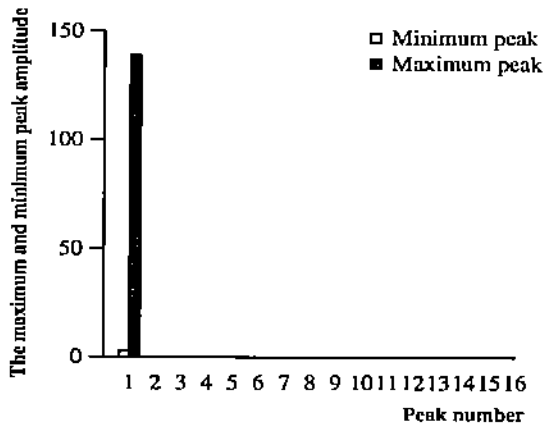
(d) COMPUTE state: time correlation



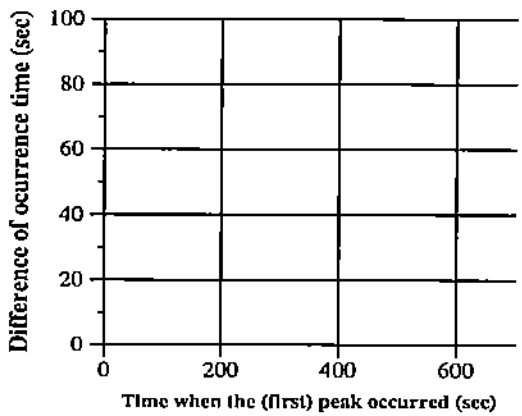
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

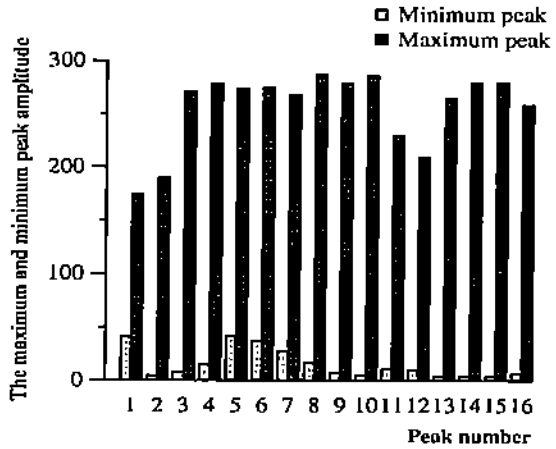


(c) COMMUN. state: amplitude correlation

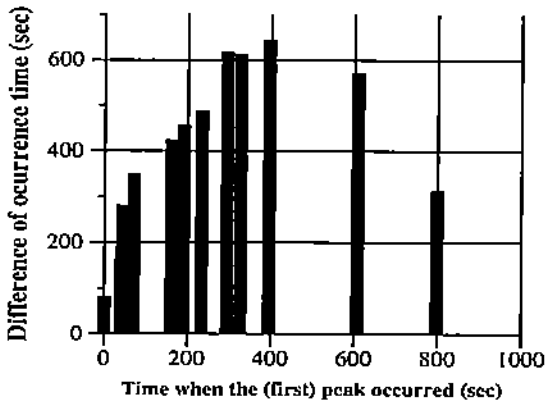


(f) COMMUN. state: time correlation

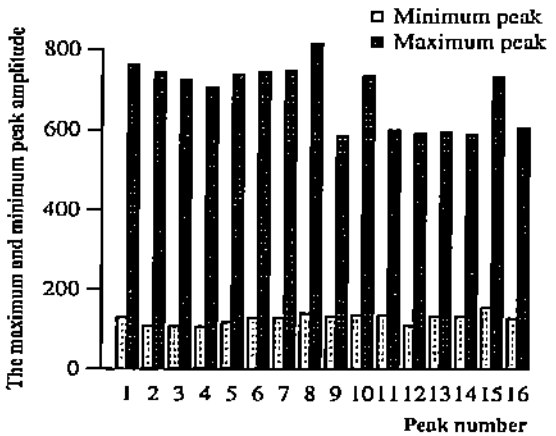
Figure 19: The page fault analysis. The correlation of amplitude and time of occurrence for the peaks of page fault activity for the Envelope (DFS mode) program running in 64 nodes.



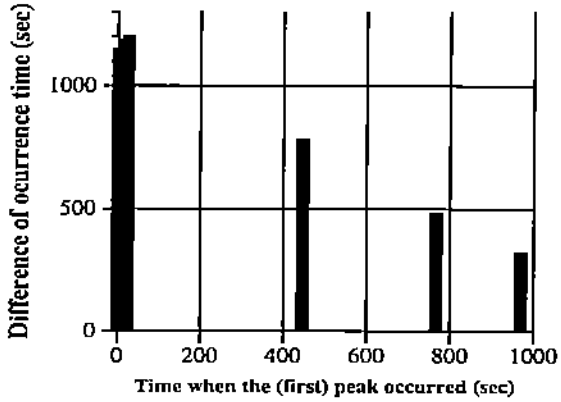
(a) COMPUTE state: amplitude correlation



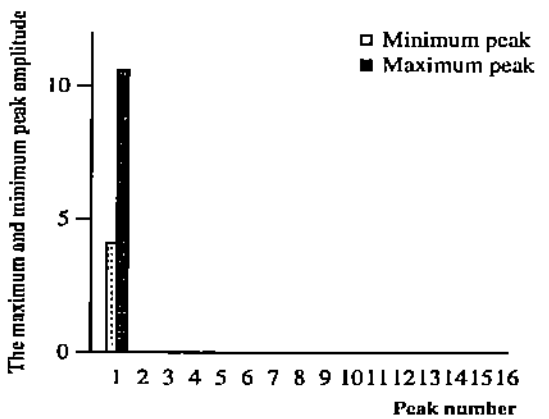
(d) COMPUTE state: time correlation



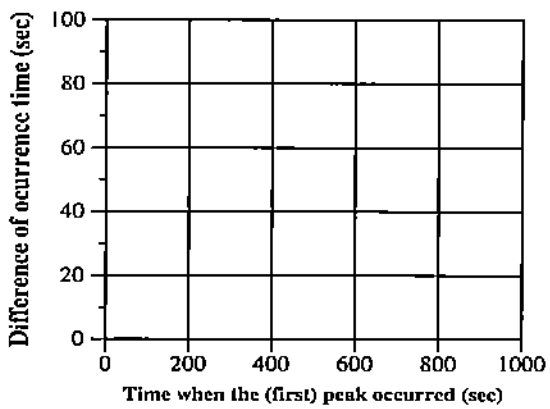
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

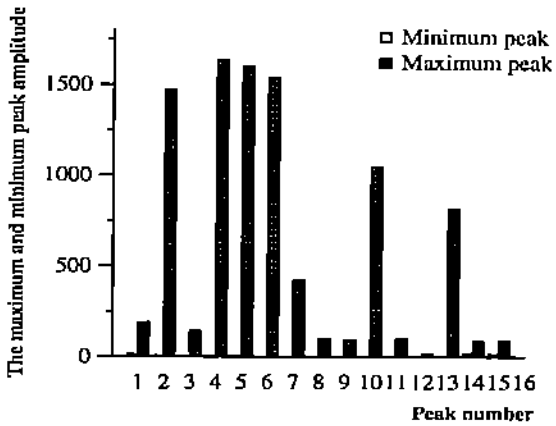


(c) COMMUN. state: amplitude correlation

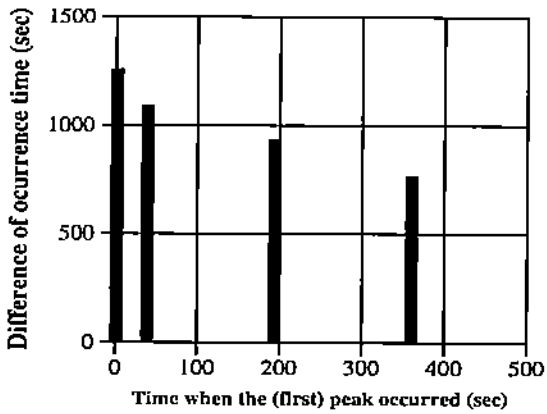


(f) COMMUN. state: time correlation

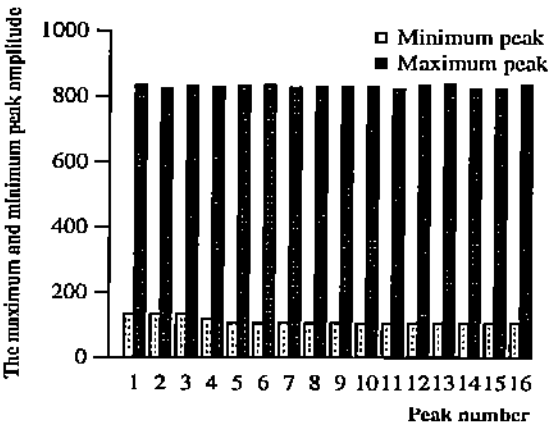
Figure 20: The page-in analysis. The correlation of amplitude and time of occurrence for the peaks of page-in activity for the Envelope (DFS mode) program running in 64 nodes.



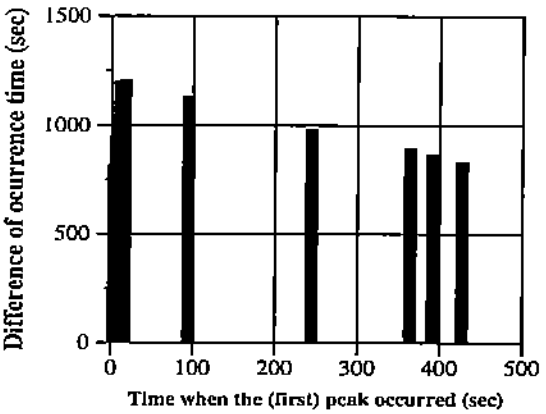
(a) COMPUTE state: amplitude correlation



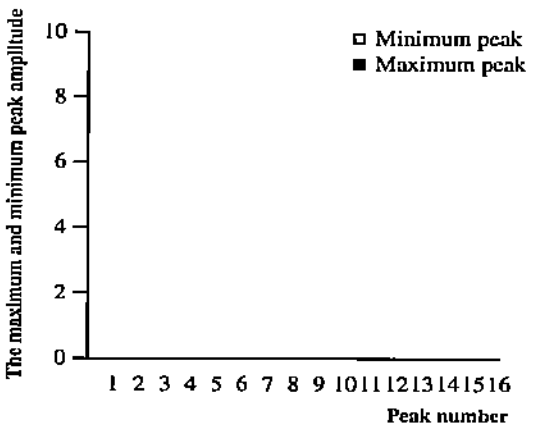
(d) COMPUTE state: time correlation



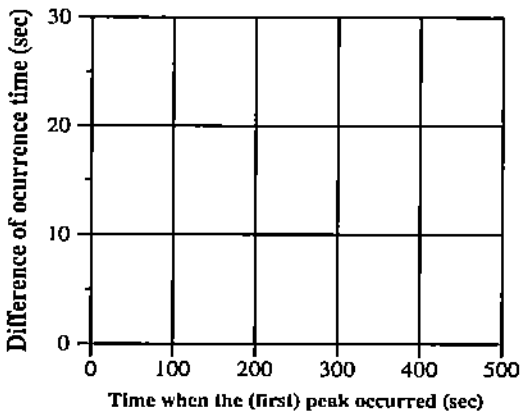
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

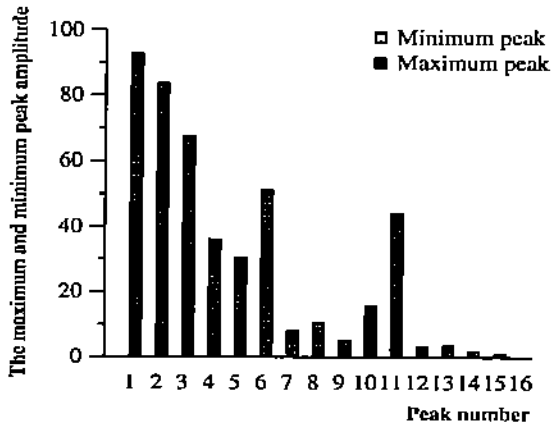


(c) COMMUN. state: amplitude correlation

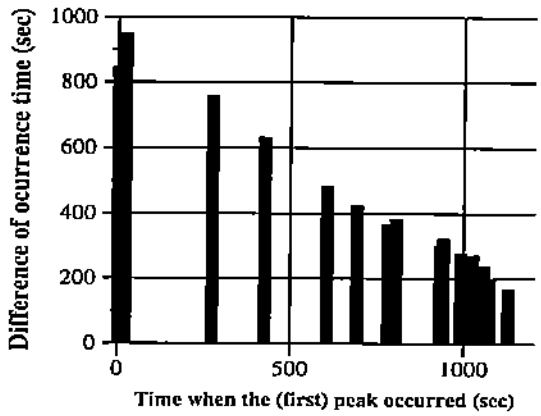


(f) COMMUN. state: time correlation

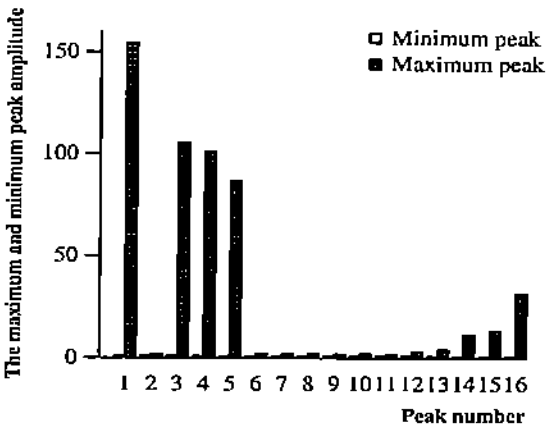
Figure 21: The copy-on-write fault analysis. The correlation of amplitude and time of occurrence for the peaks of cow fault activity for the Envelope (DFS mode) program running in 64 nodes.



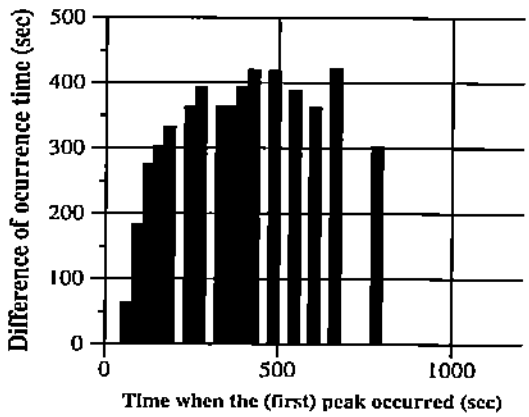
(a) COMPUTE state: amplitude correlation



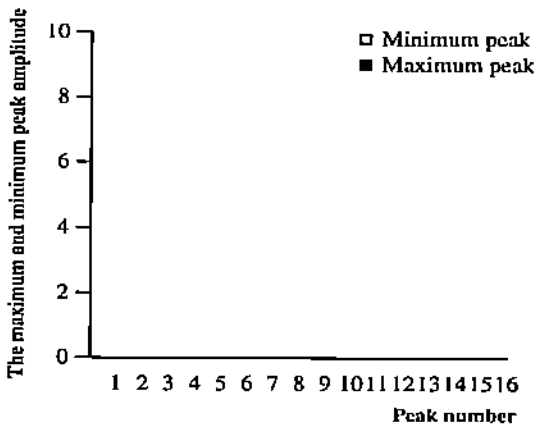
(d) COMPUTE state: time correlation



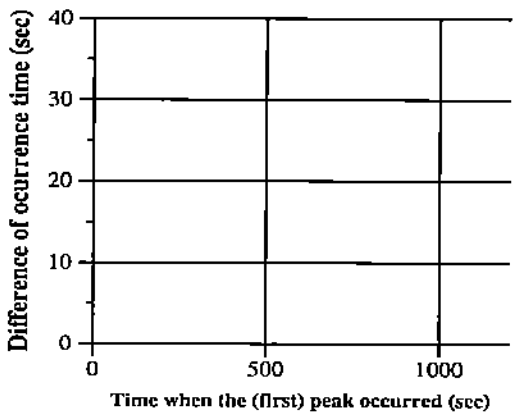
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

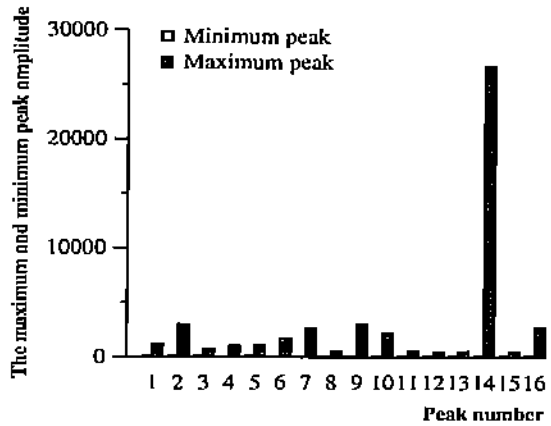


(c) COMMUN. state: amplitude correlation

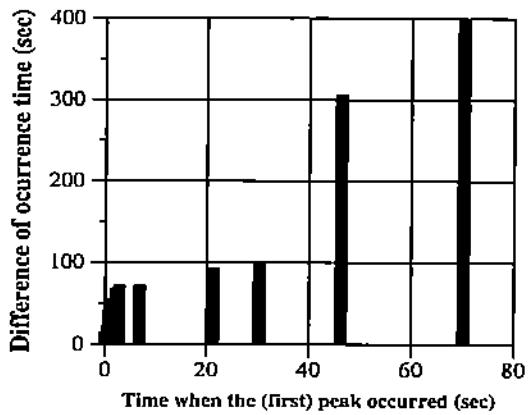


(f) COMMUN. state: time correlation

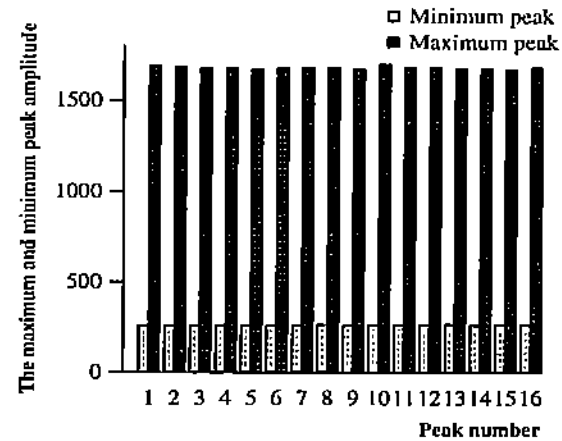
Figure 22: The page-out analysis. The correlation of amplitude and time of occurrence for the peaks of page-out activity for the Envelope (DFS mode) program running in 64 nodes.



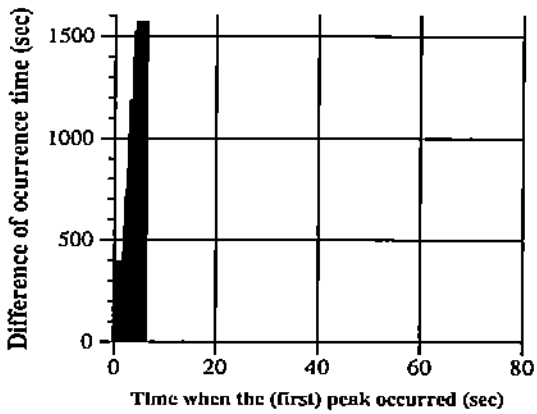
(a) COMPUTE state: amplitude correlation



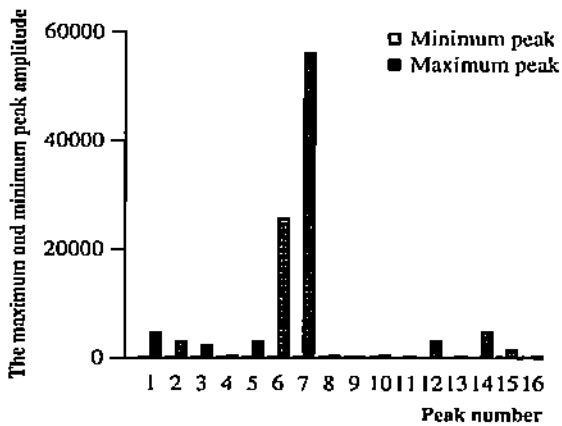
(d) COMPUTE state: time correlation



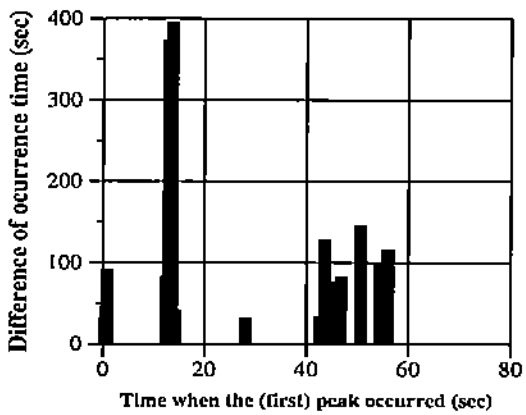
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

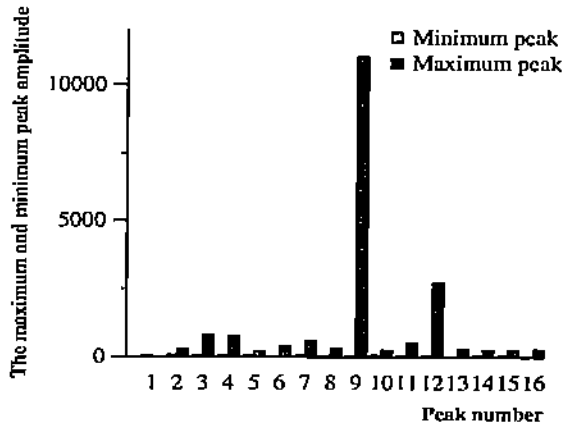


(c) COMMUN. state: amplitude correlation

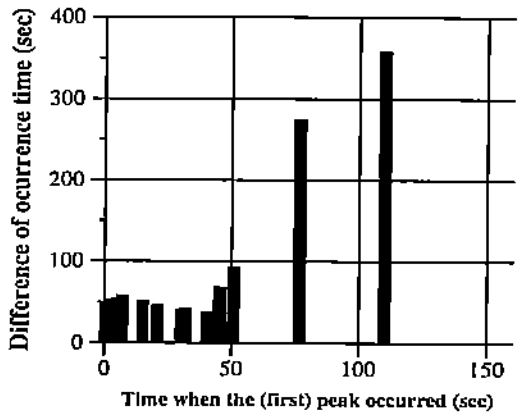


(f) COMMUN. state: time correlation

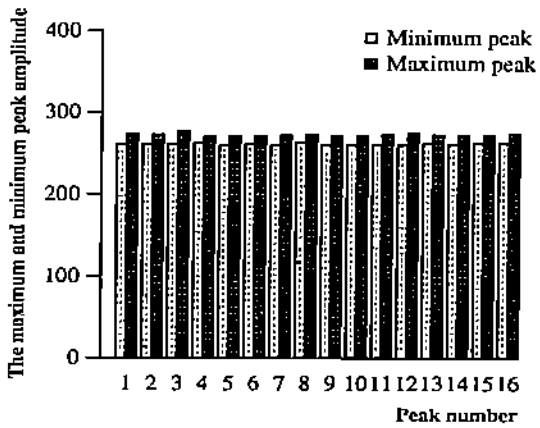
Figure 23: The page fault analysis. The correlation of amplitude and time of occurrence for the peaks of page fault activity for the Envelope (DAN mode) program running in 16 nodes.



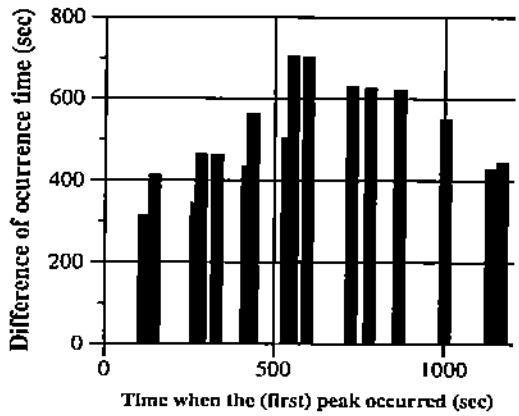
(a) COMPUTE state: amplitude correlation



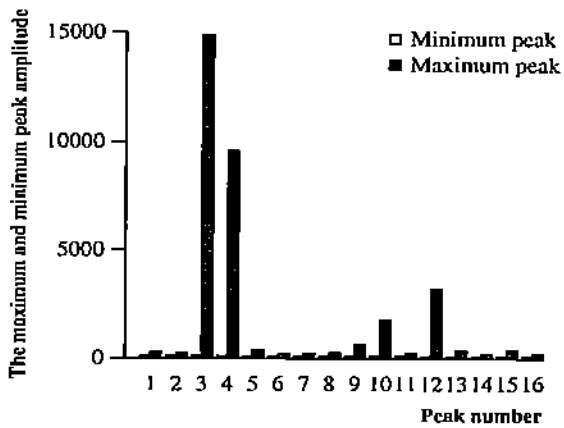
(d) COMPUTE state: time correlation



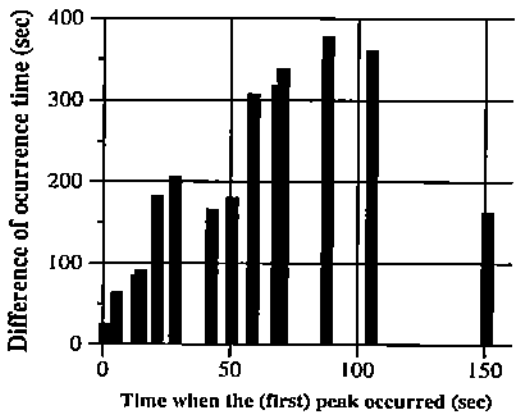
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

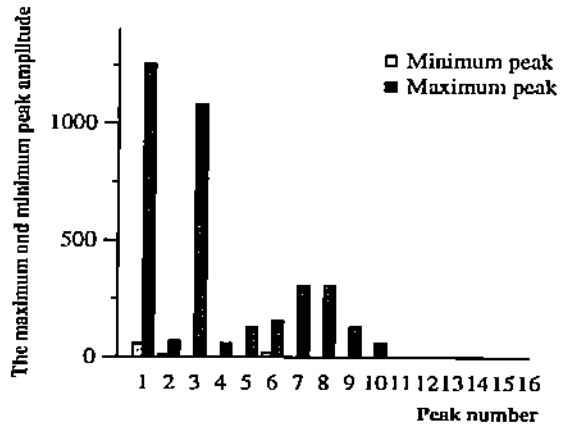


(c) COMMUN. state: amplitude correlation

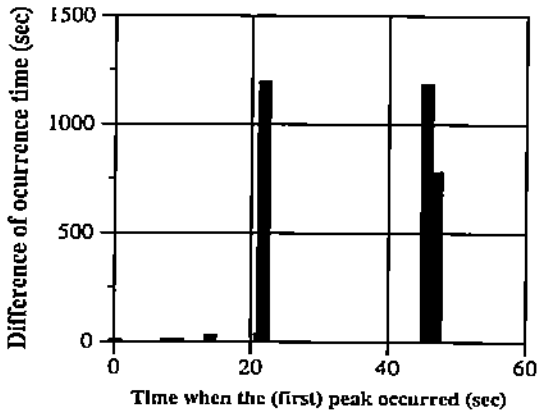


(f) COMMUN. state: time correlation

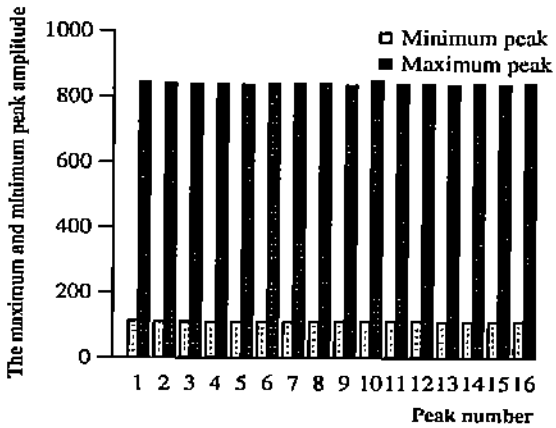
Figure 24: The page-in analysis. The correlation of amplitude and time of occurrence for the peaks of page-in activity for the Envelope (DAN mode) program running in 16 nodes.



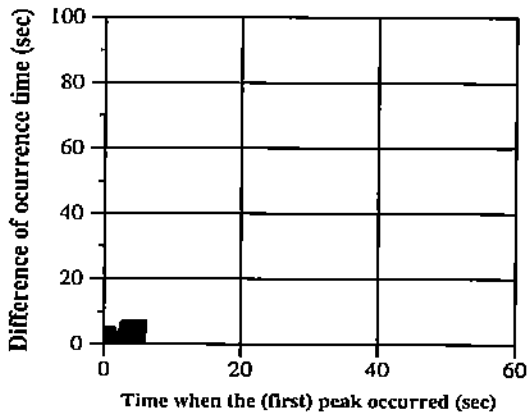
(a) COMPUTE state: amplitude correlation



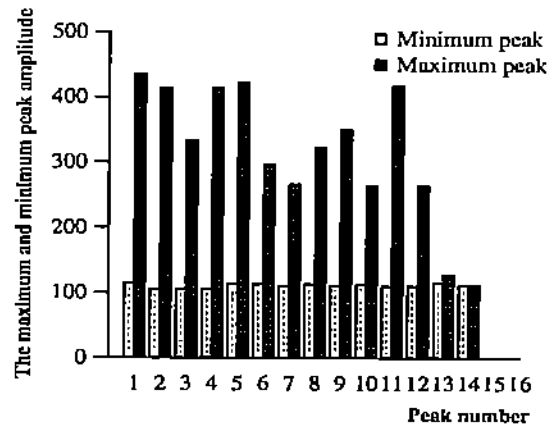
(d) COMPUTE state: time correlation



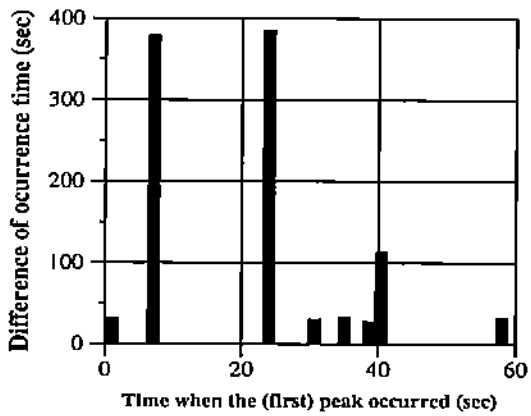
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

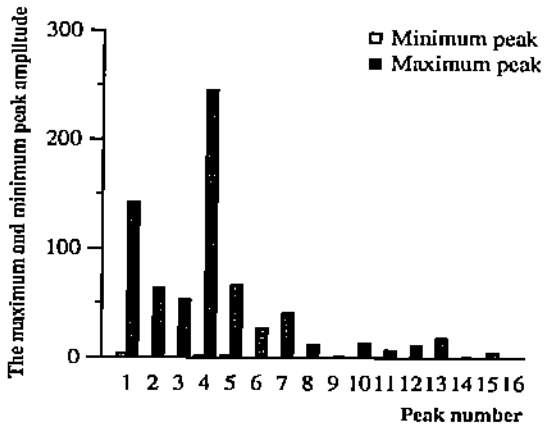


(c) COMMUN. state: amplitude correlation

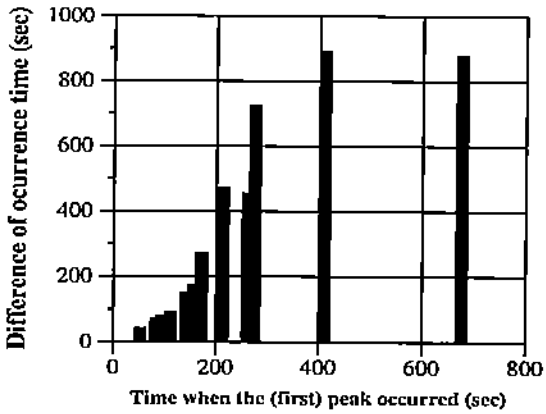


(f) COMMUN. state: time correlation

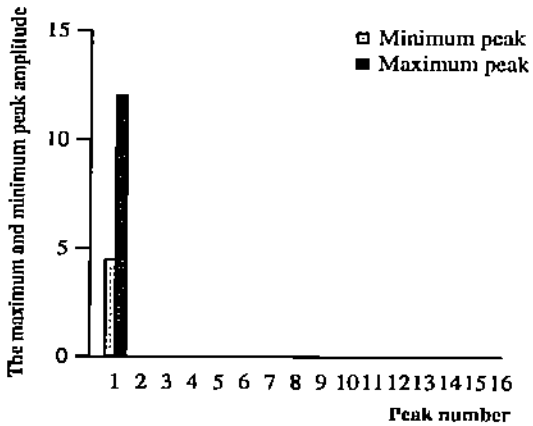
Figure 25: The copy-on-write fault analysis. The correlation of amplitude and time of occurrence for the peaks of cow fault activity for the Envelope (DAN mode) program running in 16 nodes.



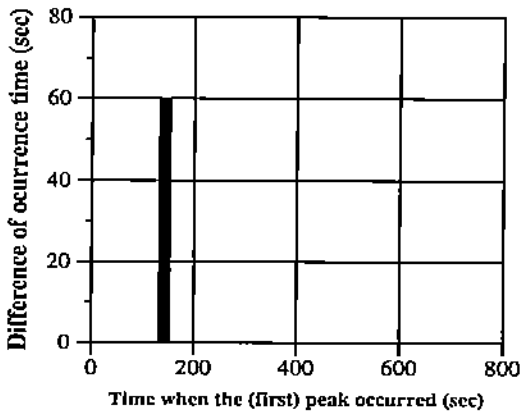
(a) COMPUTE state: amplitude correlation



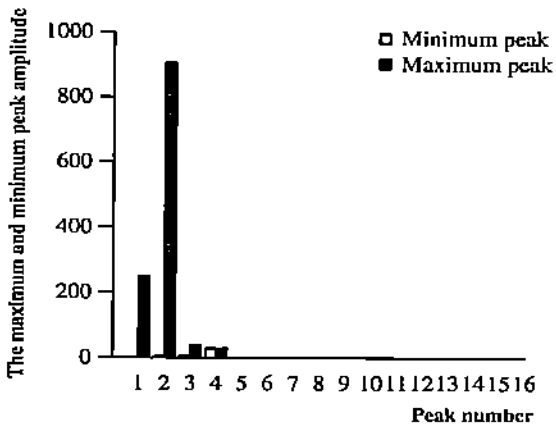
(d) COMPUTE state: time correlation



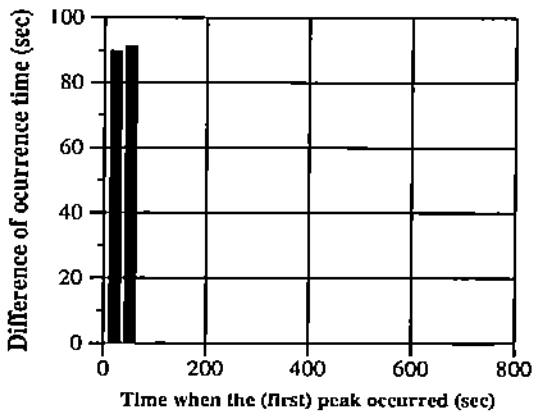
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

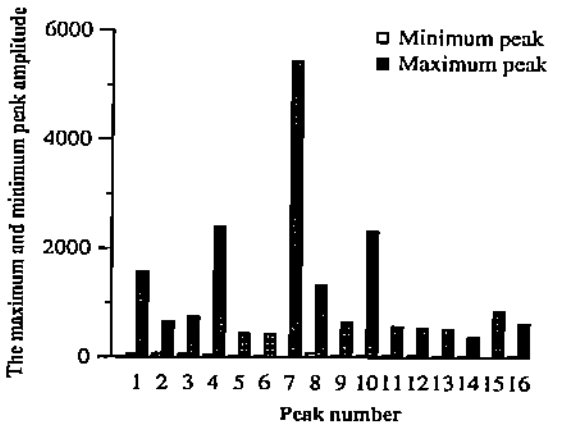


(c) COMMUN. state: amplitude correlation

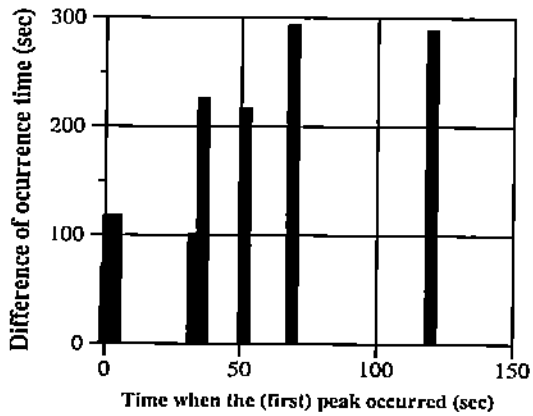


(f) COMMUN. state: time correlation

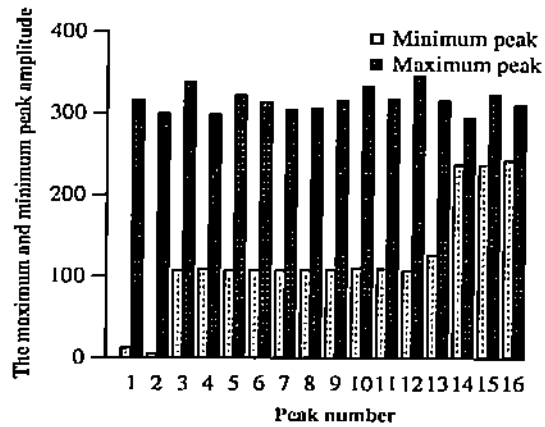
Figure 26: The page-out analysis. The correlation of amplitude and time of occurrence for the peaks of page-out activity for the Envelope (DAN mode) program running in 16 nodes.



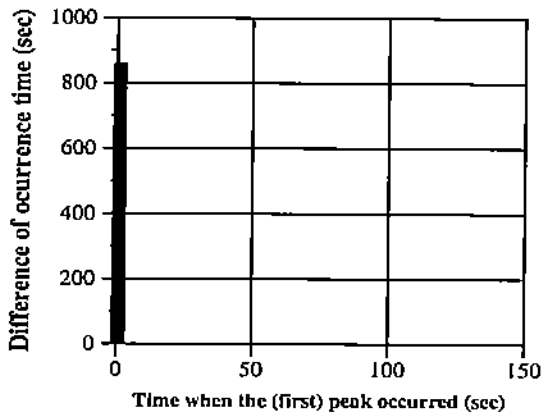
(a) COMPUTE state: amplitude correlation



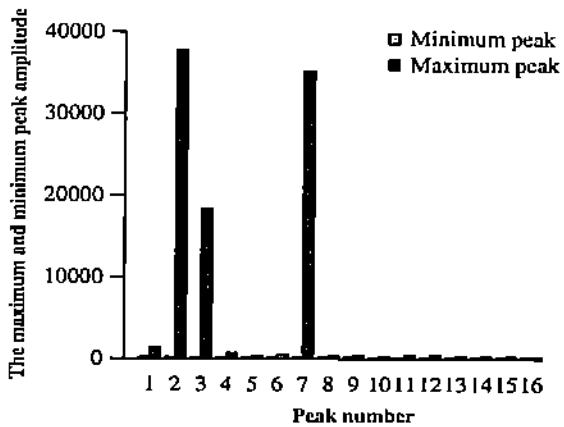
(d) COMPUTE state: time correlation



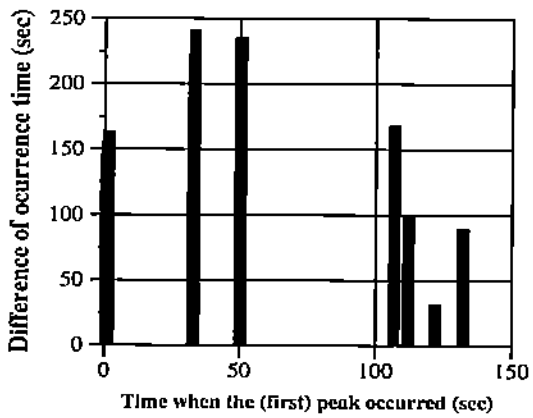
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

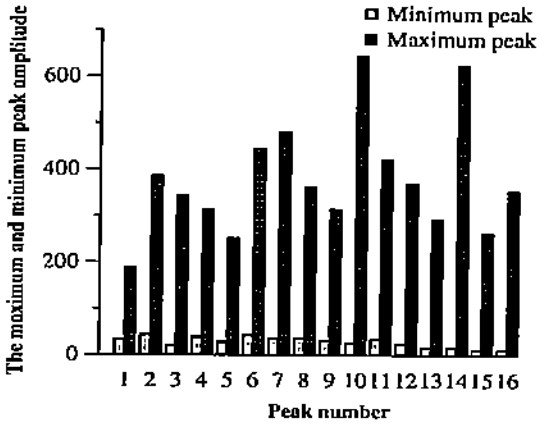


(c) COMMUN. state: amplitude correlation

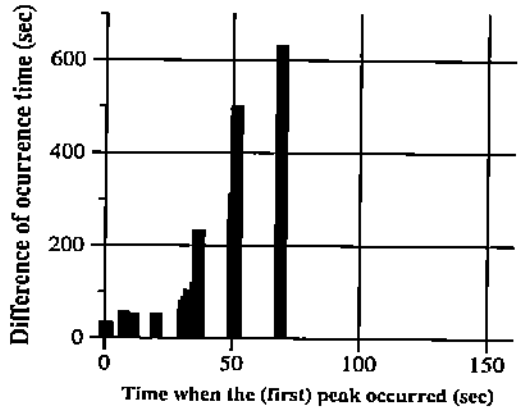


(f) COMMUN. state: time correlation

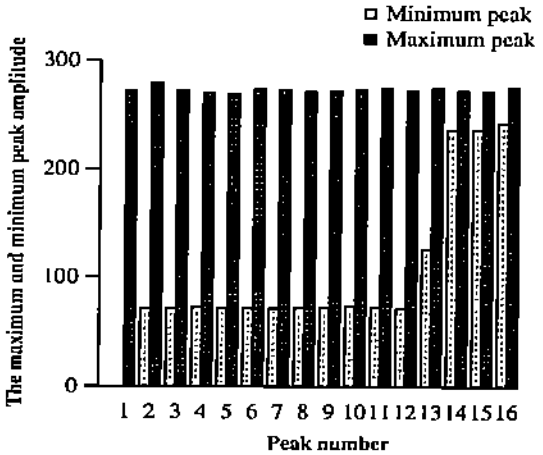
Figure 27: The page fault analysis. The correlation of amplitude and time of occurrence for the peaks of page fault activity for the Envelope (DAN mode) program running in 32 nodes.



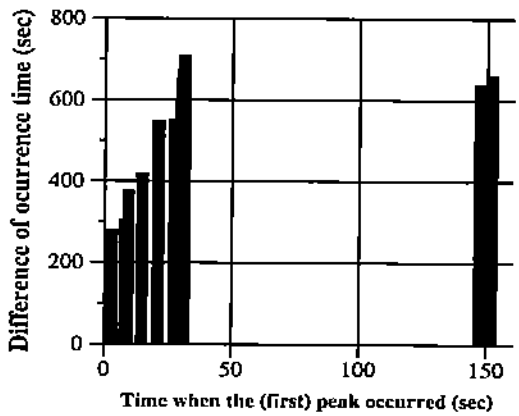
(a) COMPUTE state: amplitude correlation



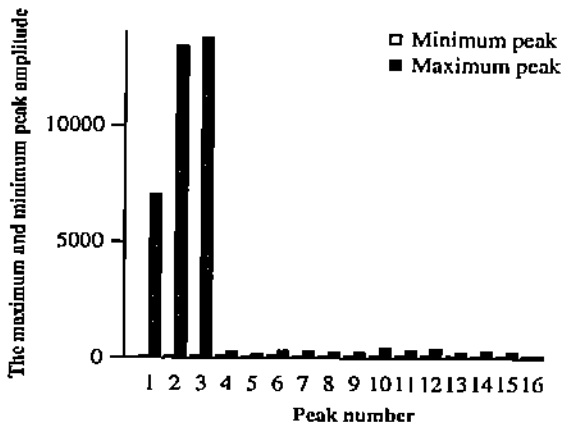
(d) COMPUTE state: time correlation



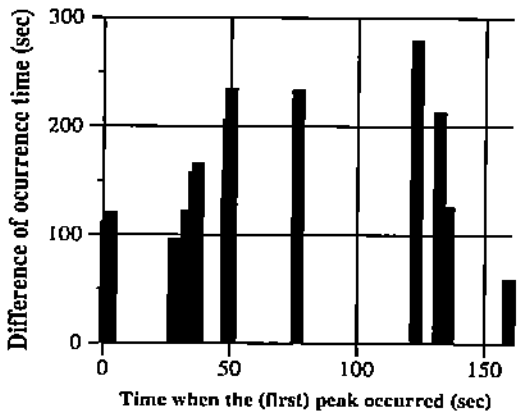
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

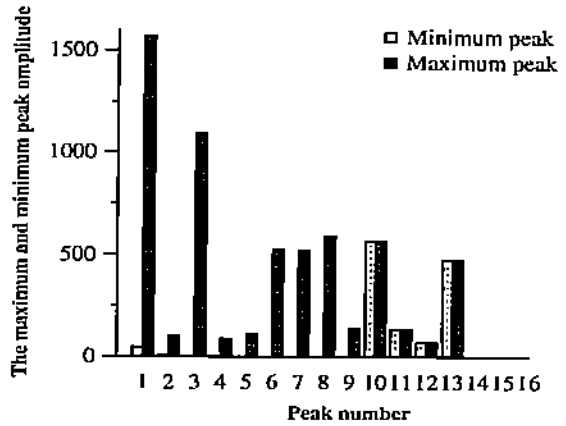


(c) COMMUN. state: amplitude correlation

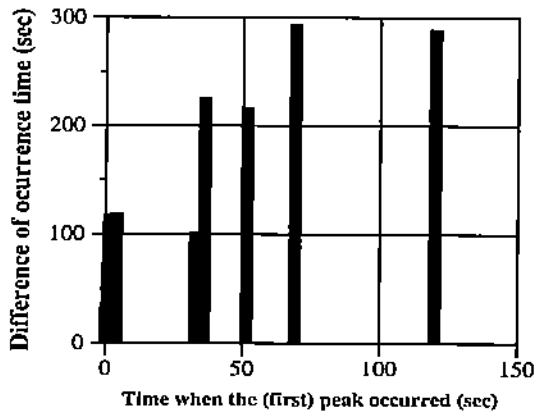


(f) COMMUN. state: time correlation

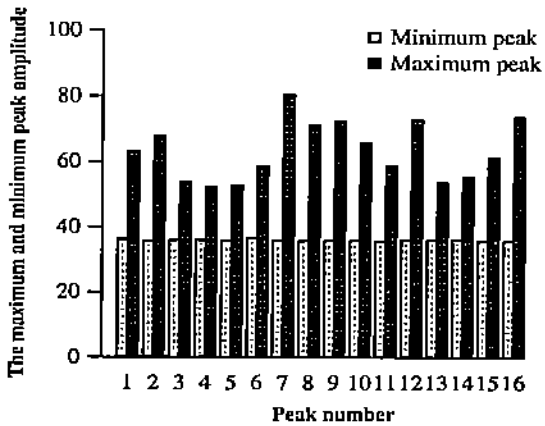
Figure 28: The page-in analysis. The correlation of amplitude and time of occurrence for the peaks of page-in activity for the Envelope (DAN mode) program running in 32 nodes.



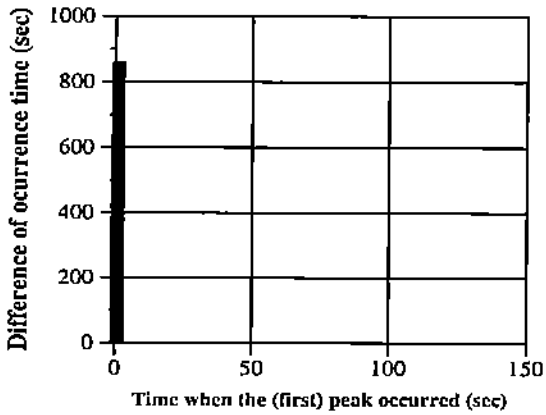
(a) COMPUTE state: amplitude correlation



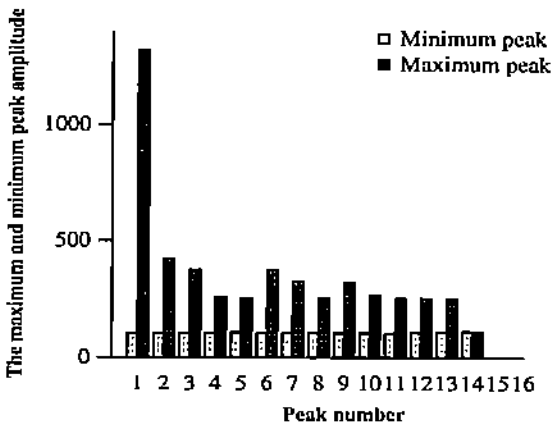
(d) COMPUTE state: time correlation



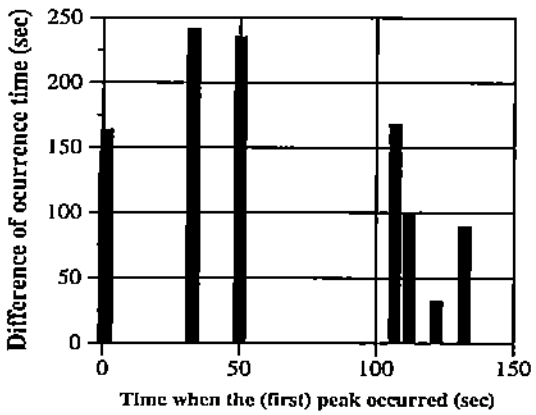
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

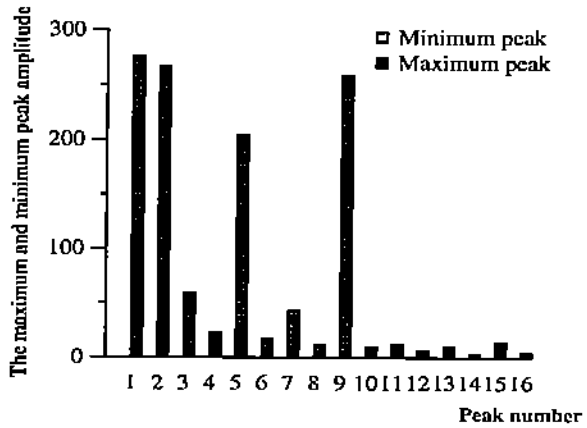


(c) COMMUN. state: amplitude correlation

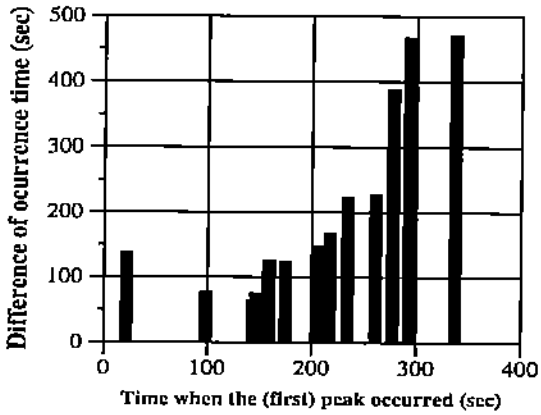


(f) COMMUN. state: time correlation

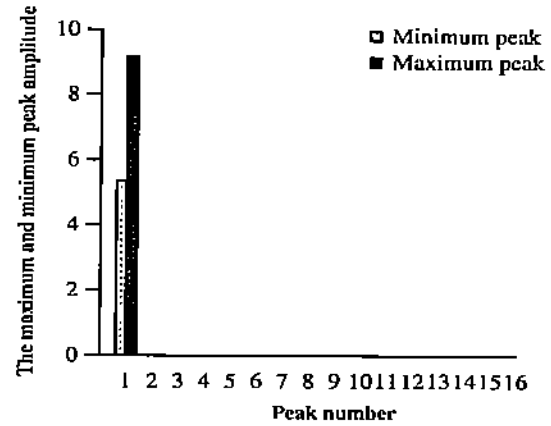
Figure 29: The copy-on-write fault analysis. The correlation of amplitude and time of occurrence for the peaks of cow fault activity for the Envelope (DAN mode) program running in 32 nodes.



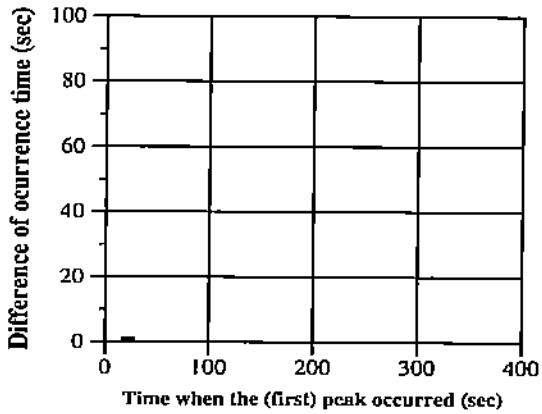
(a) COMPUTE state: amplitude correlation



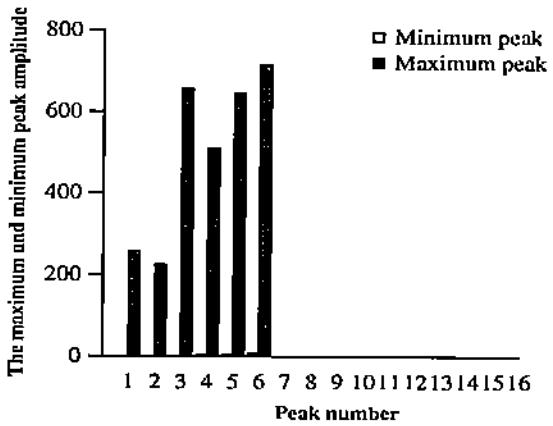
(d) COMPUTE state: time correlation



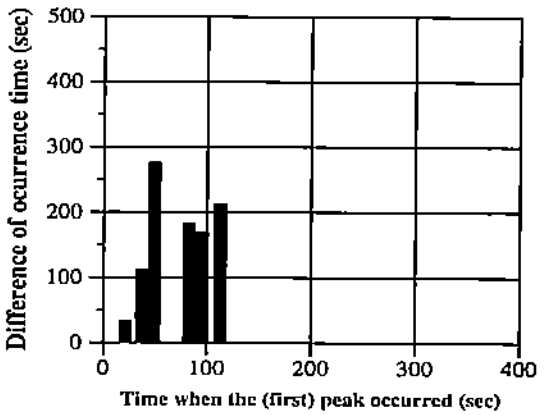
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

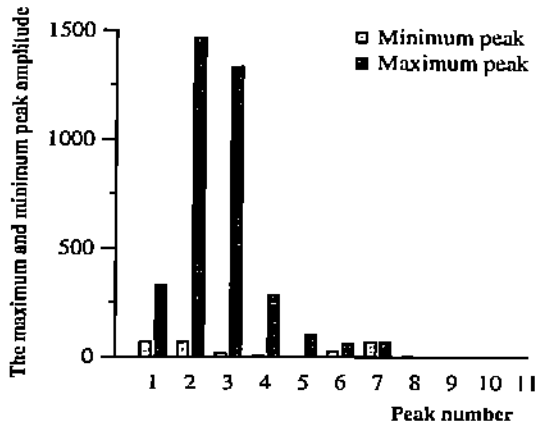


(c) COMMUN. state: amplitude correlation

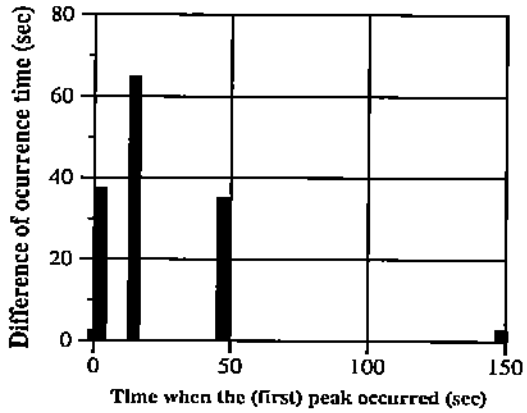


(f) COMMUN. state: time correlation

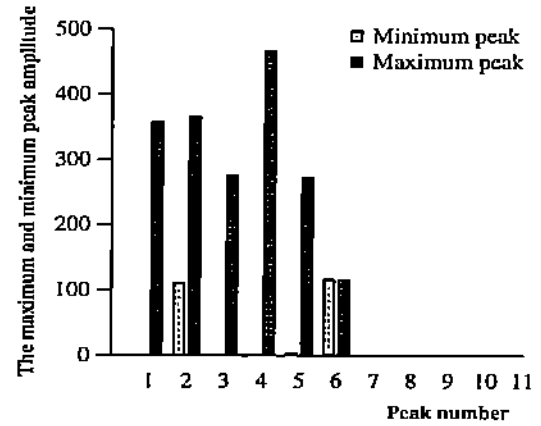
Figure 30: The page-out analysis. The correlation of amplitude and time of occurrence for the peaks of page-out activity for the Envelope (DAN mode) program running in 32 nodes.



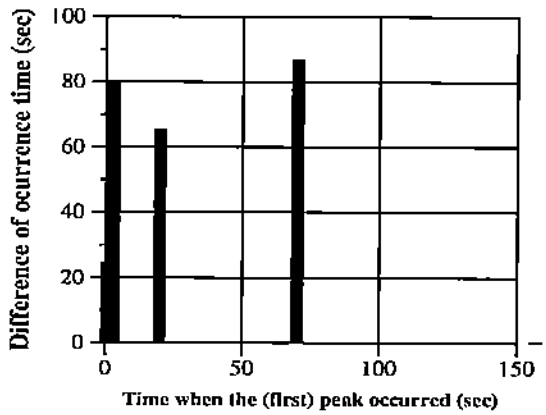
(a) COMPUTE state: amplitude correlation



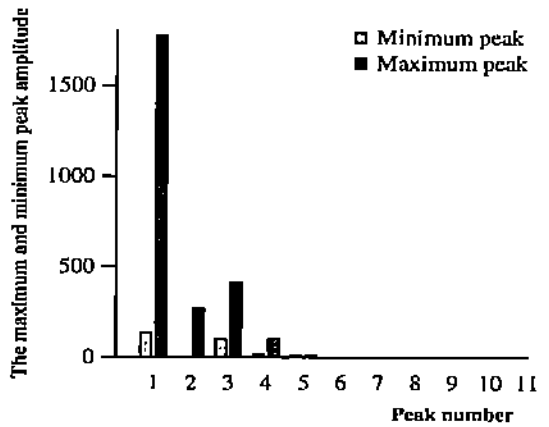
(d) COMPUTE state: time correlation



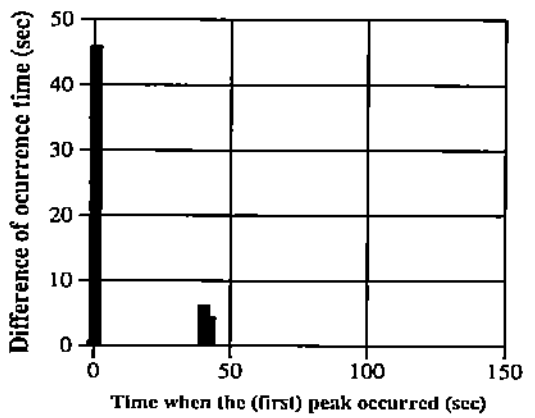
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

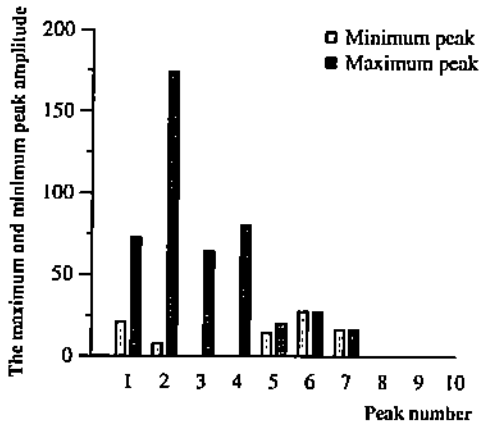


(c) COMMUN. state: amplitude correlation

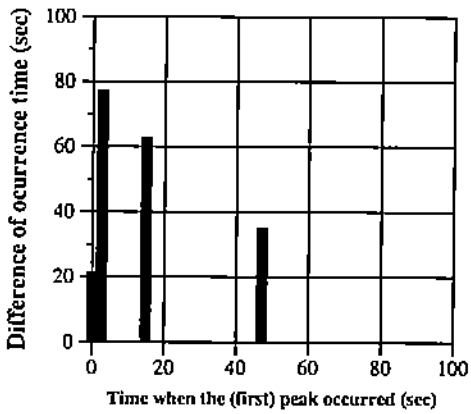


(f) COMMUN. state: time correlation

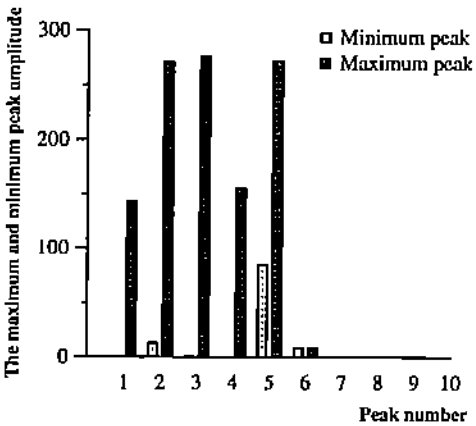
Figure 31: The page fault analysis. The correlation of amplitude and time of occurrence for the peaks of page fault activity for the FFTSynth program running in 8 nodes.



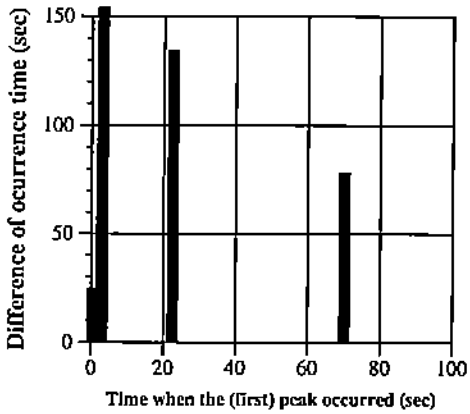
(a) COMPUTE state: amplitude correlation



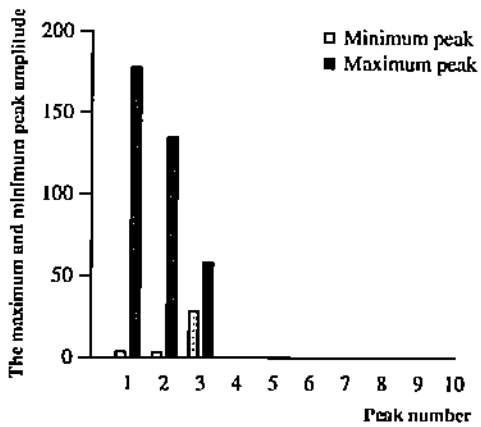
(d) COMPUTE state: time correlation



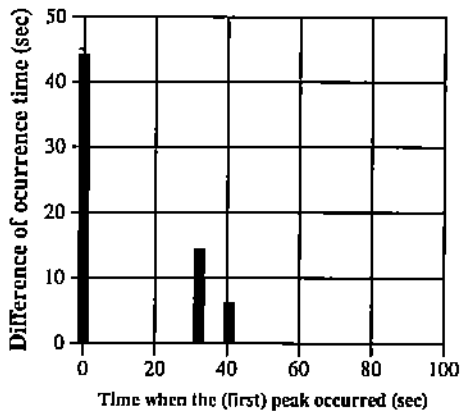
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

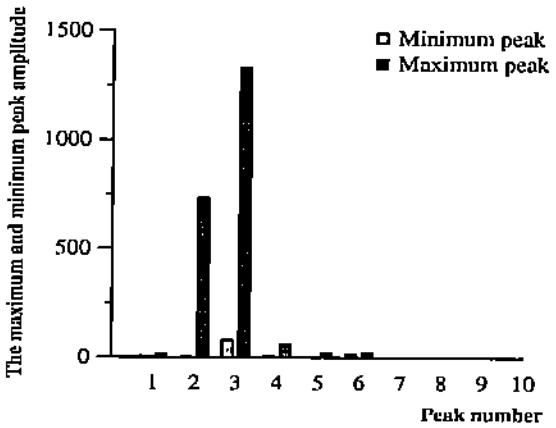


(c) COMMUN. state: amplitude correlation

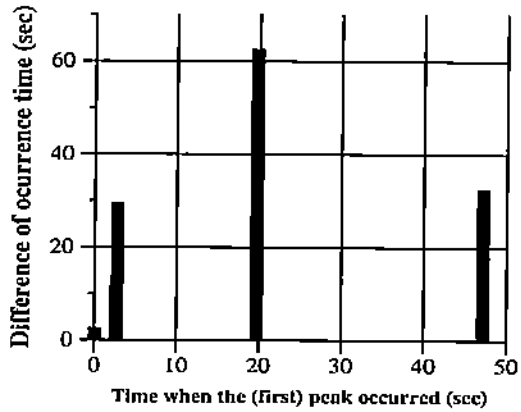


(f) COMMUN. state: time correlation

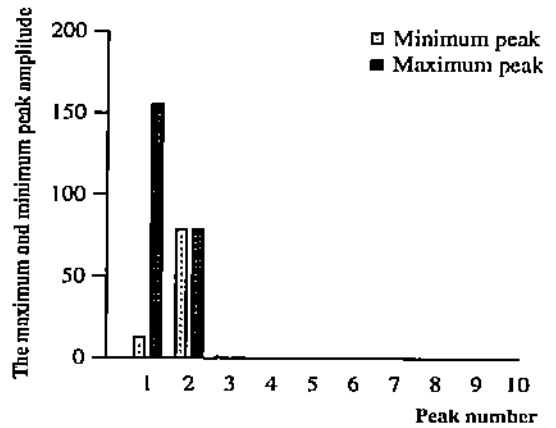
Figure 32: The page-in analysis. The correlation of amplitude and time of occurrence for the peaks of page-in activity for the FFTSynth program running in 8 nodes.



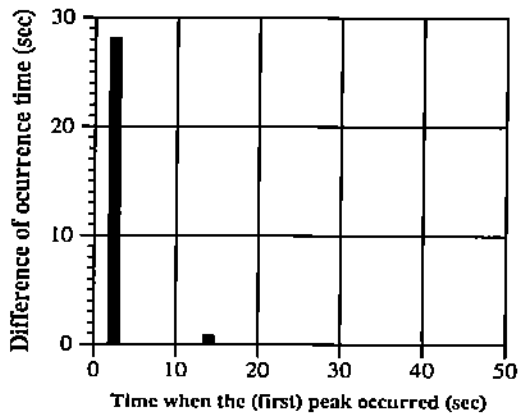
(a) COMPUTE state: amplitude correlation



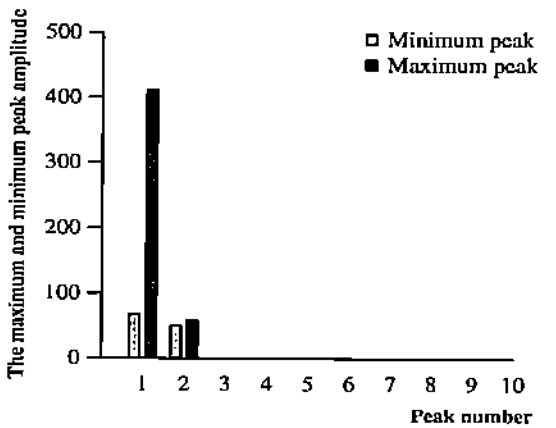
(d) COMPUTE state: time correlation



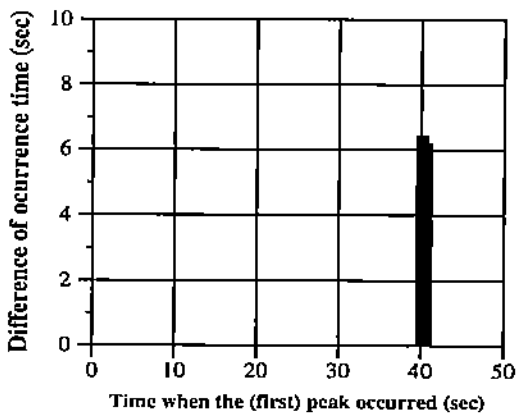
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

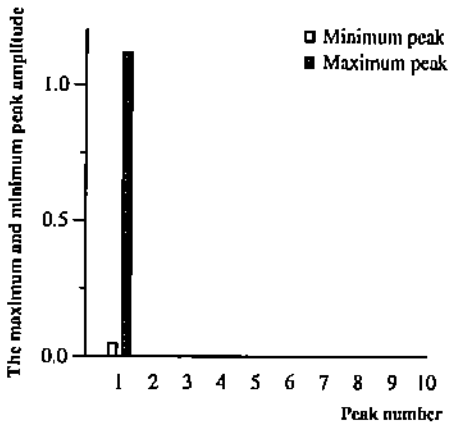


(c) COMMUN. state: amplitude correlation

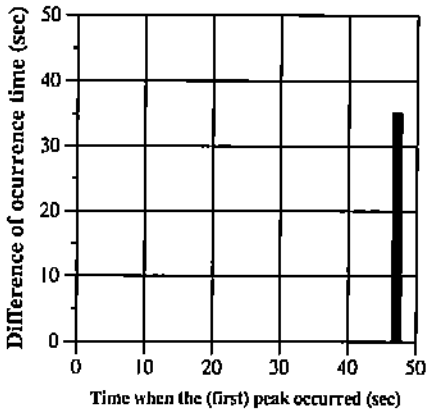


(f) COMMUN. state: time correlation

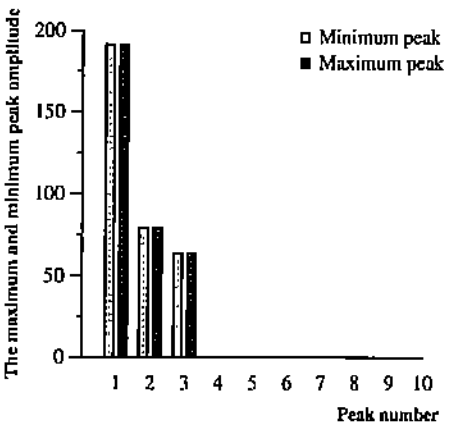
Figure 33: The copy-on-write fault analysis. The correlation of amplitude and time of occurrence for the peaks of cow fault activity for the FFTSynth program running in 8 nodes.



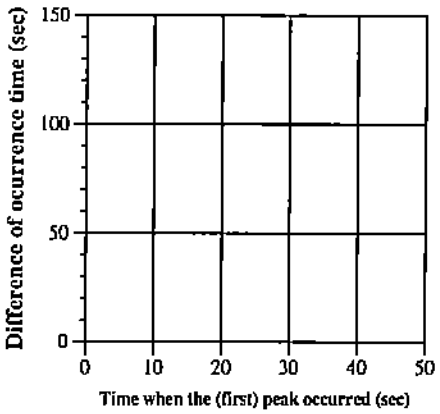
(a) COMPUTE state: amplitude correlation



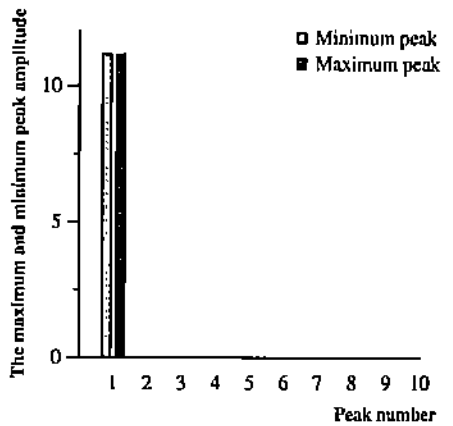
(d) COMPUTE state: time correlation



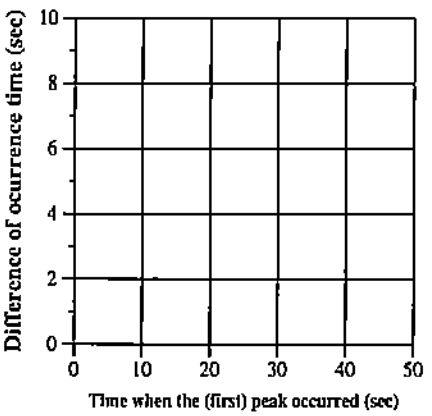
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

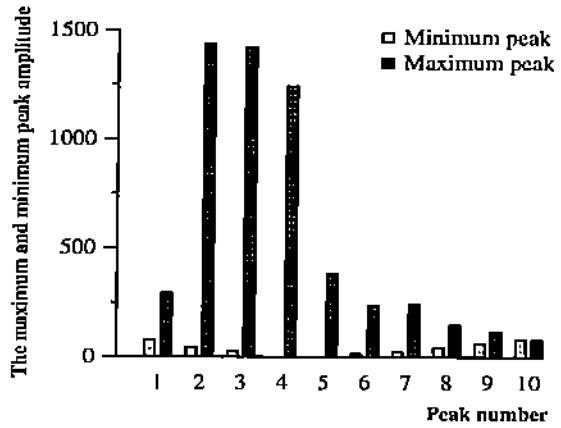


(c) COMMUN. state: amplitude correlation

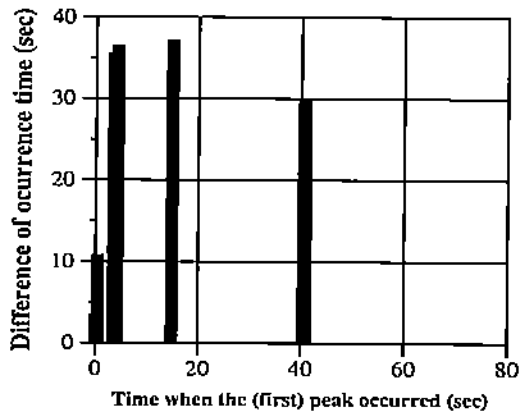


(f) COMMUN. state: time correlation

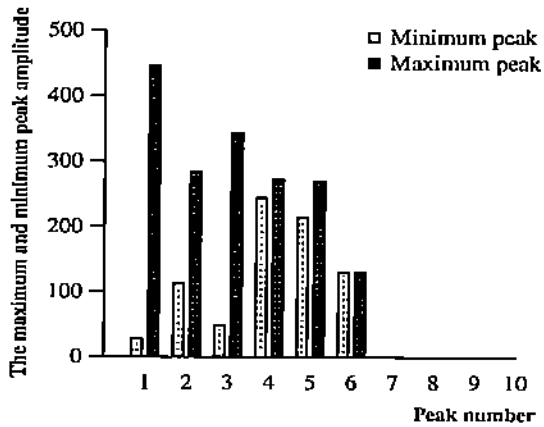
Figure 34: The page-out analysis. The correlation of amplitude and time of occurrence for the peaks of page-out activity for the FFTSynth program running in 8 nodes.



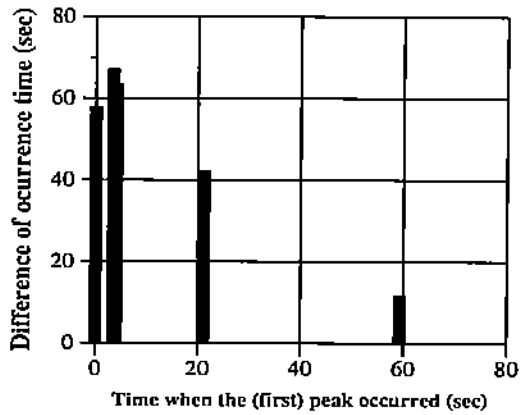
(a) COMPUTE state: amplitude correlation



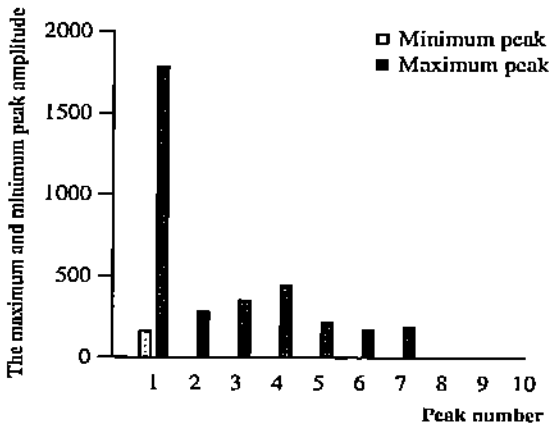
(d) COMPUTE state: time correlation



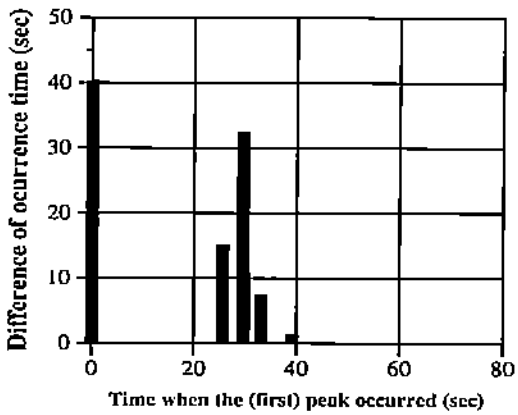
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

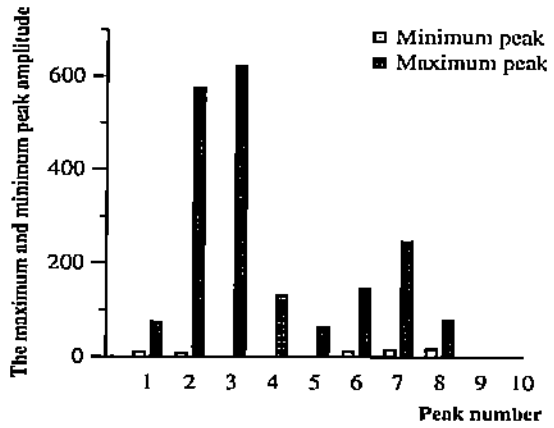


(c) COMMUN. state: amplitude correlation

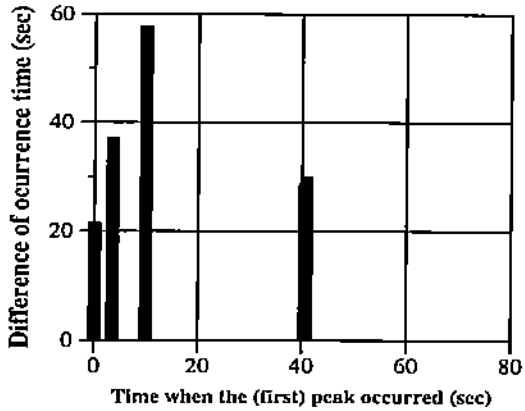


(f) COMMUN. state: time correlation

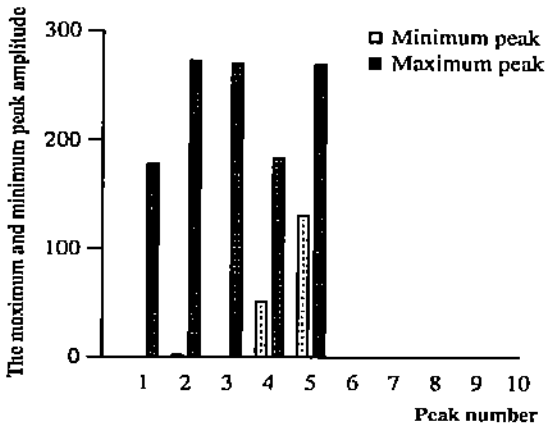
Figure 35: The page fault analysis. The correlation of amplitude and time of occurrence for the peaks of page fault activity for the FFTSynth program running in 16 nodes.



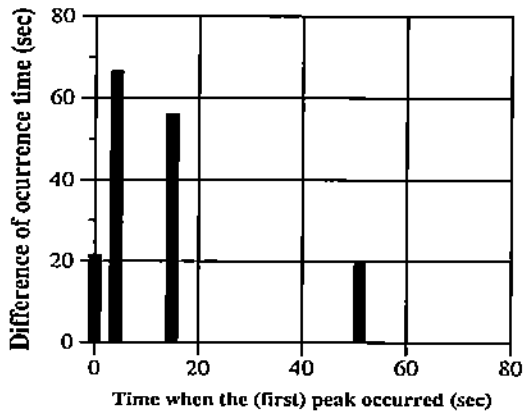
(a) COMPUTE state: amplitude correlation



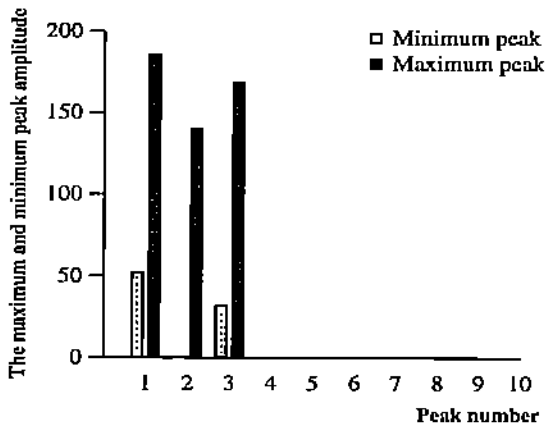
(d) COMPUTE state: time correlation



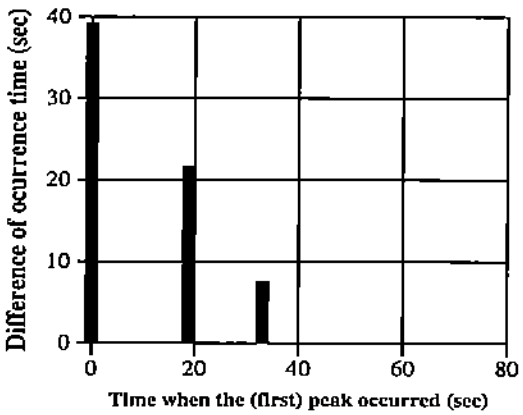
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

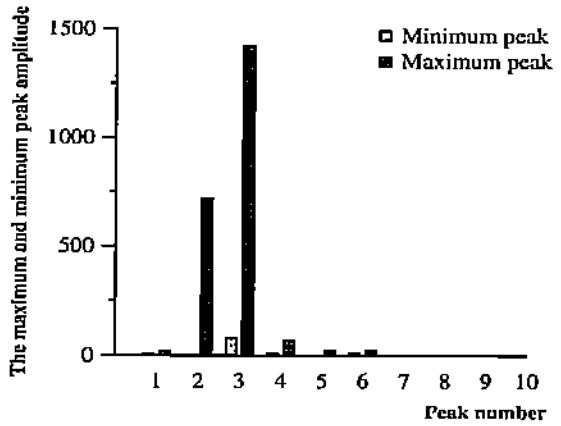


(c) COMMUN. state: amplitude correlation

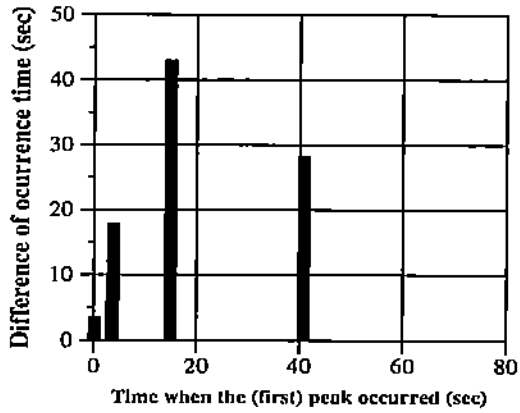


(f) COMMUN. state: time correlation

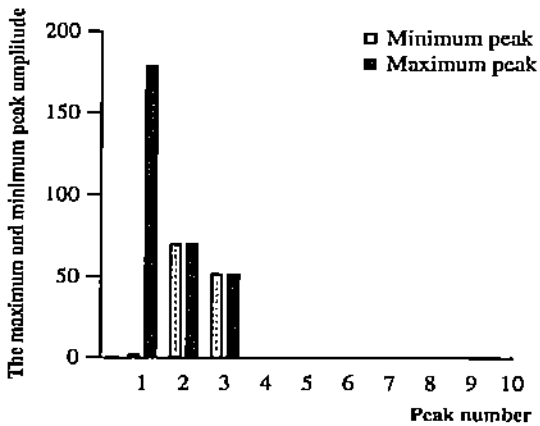
Figure 36: The page-in analysis. The correlation of amplitude and time of occurrence for the peaks of page-in activity for the FFTSynth program running in 16 nodes.



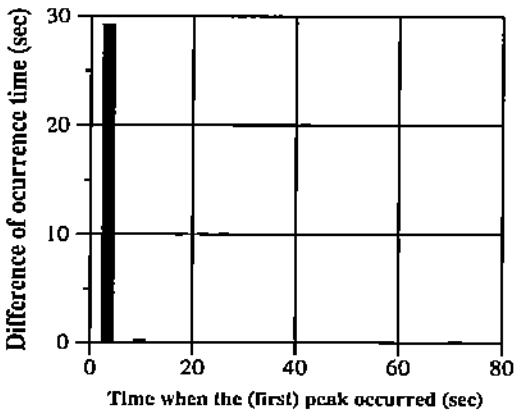
(a) COMPUTE state: amplitude correlation



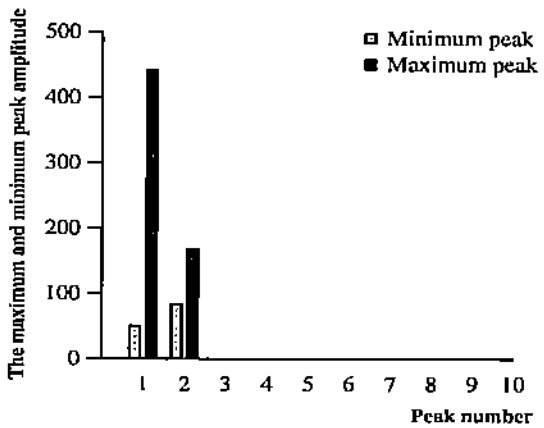
(d) COMPUTE state: time correlation



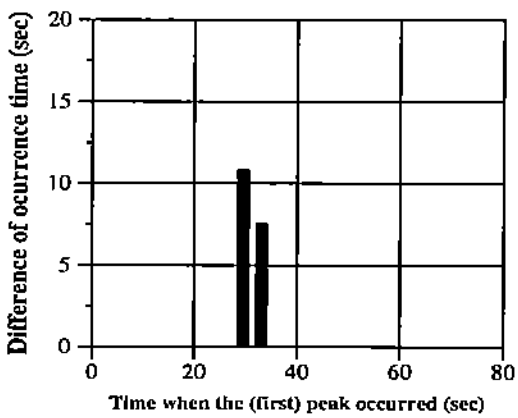
(b) I/O state: amplitude correlation



(e) I/O state: time correlation

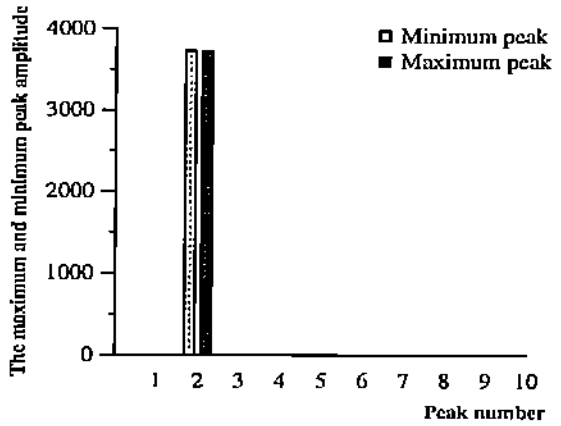


(c) COMMUN. state: amplitude correlation

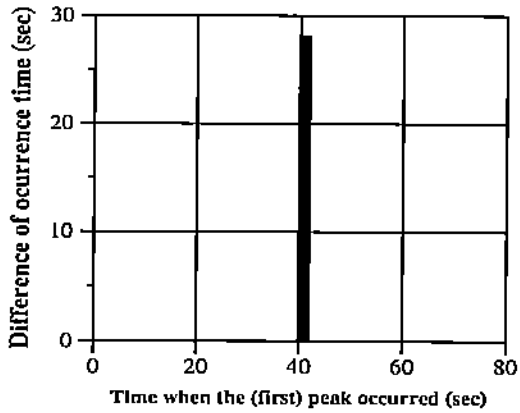


(f) COMMUN. state: time correlation

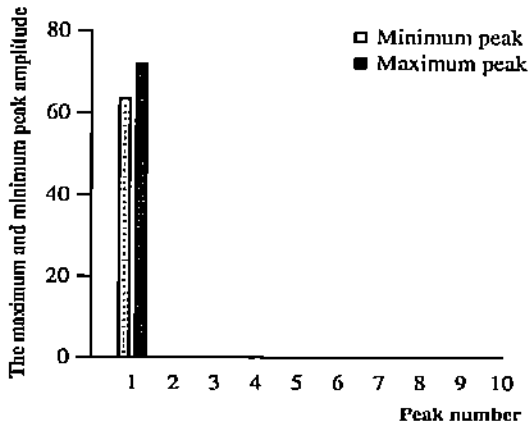
Figure 37: The copy-on-write fault analysis. The correlation of amplitude and time of occurrence for the peaks of cow fault activity for the FFTSynth program running in 16 nodes.



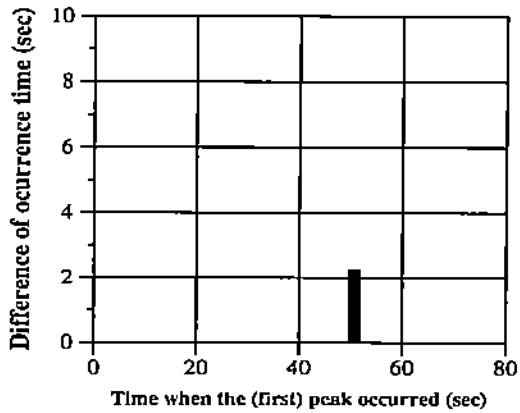
(a) COMPUTE state: amplitude correlation



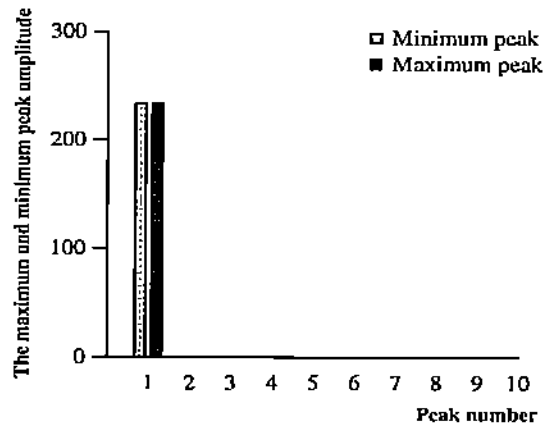
(d) COMPUTE state: time correlation



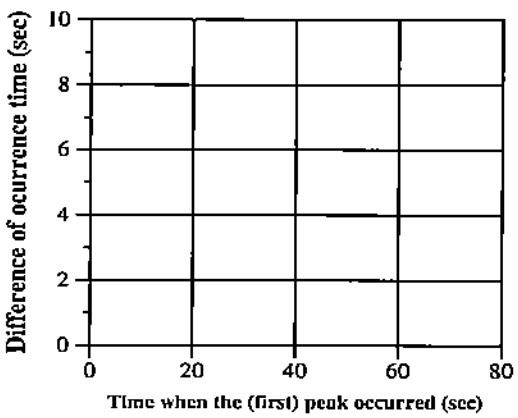
(b) I/O state: amplitude correlation



(e) I/O state: time correlation



(c) COMMUN. state: amplitude correlation



(f) COMMUN. state: time correlation

Figure 38: The page-out analysis. The correlation of amplitude and time of occurrence for the peaks of page-out activity for the FFTSynth program running in 16 nodes.